# EXCEPTION HANDLING IN OBJECT-ORIENTED ANALYSIS AND DESIGN

by

**Annelise Janse van Rensburg**

submitted in part fulfillment of the requirements
for the degree of

**MASTER OF SCIENCE**

in the subject

**INFORMATION SYSTEMS**

at the

**UNIVERSITY OF SOUTH AFRICA**

**SUPERVISOR: PROF K V RENAUD**

NOVEMBER 2001

# SUMMARY

**Title:** EXCEPTION HANDLING IN OBJECT-ORIENTED ANALYSIS AND DESIGN

**By:** Annelise Janse van Rensburg

**Degree:** M Sc

**Subject:** Information Systems

**Promoter:** Prof. K V Renaud

**Summary:** This dissertation investigates current trends concerning exceptions. Exceptions influence the reliability of software systems. In order to develop software systems that are more robust, thus delivering higher availability at a lower development and operating cost, the occurrence of exceptions needs to be reduced and the effects of the exceptions controlled. In order to do this, issues such as detection, identification, classification, propagation, handling, language implementation, software testing and reporting of exceptions must be attended to. Although some of these areas are well researched there are remaining problems. The quest is to establish if a unified exception-handling framework is possible and viable, which can address the issues and problems throughout the software development life cycle, and if so, the requirements for such a framework.

**Keywords:** Exception; Detection; Identification; Classification; Propagation; Exception Handling; Object-Oriented; Testing; Reporting; Software Development.

## STATEMENT OF ORIGINALITY OF DISSERTATION

I declare that
EXCEPTION HANDLING IN OBJECT-ORIENTED ANALYSIS AND DESIGN
is my own work and that all the sources that I have used or quoted have been indicated and acknowledged by means of complete references.

# CONTENTS

**Acknowledgements** .......... v

**Abstract & Keywords** .......... vi

**Glossary** .......... vii

**1 Introduction** .......... 1

1.1 Motivation for the Research .......... 1

1.2 Application of the Research .......... 1

1.3 Scope of the Dissertation .......... 2

1.4 Structure of the Dissertation .......... 3

**2 Being Realistic About Exceptions** .......... 5

2.1 Introduction .......... 5

2.2 The Causes and Origins of Exceptions .......... 6

2.3 Software Testing .......... 8

2.4 Testing Techniques .......... 11

2.4.1 Collaboration Techniques .......... 11

2.4.1.1 Dependability Cases .......... 11

2.4.1.2 Reviews .......... 13

2.4.2 Execution-Style Techniques .......... 14

2.4.2.1 Balista Robustness Testing .......... 14

2.4.2.2 Black Box / Functional Testing .......... 14

2.4.2.3 White Box Testing .......... 14

2.4.3 Multi-Version Techniques .......... 15

2.4.3.1 N-Version Programming .......... 15

2.4.3.2 Recovery Blocks .......... 15

2.4.4 Mathematical Techniques .......... 16

2.4.4.1 Correctness Proofs .......... 16

2.4.5 Summary of the Testing Techniques .......... 16

2.5 Categorising Results .......... 18

2.6 Summary .......... 20

**3 Background ........ 21**

3.1 Introduction ........ 21

3.2 Exception handling in the Phases of the Software Development Life Cycle ........ 23

3.3 General Policies regarding Exception Handling ........ 25

3.3.1 Ignore Policy ........ 26

3.3.2 Acknowledge-Notify Policy ........ 26

3.3.3 React-Repair Policy ........ 27

3.3.4 Hybrid Policies ........ 28

3.4 Strategies for Policies ........ 28

3.4.1 Detection of Exceptions ........ 29

3.4.1.1 Violation of Constraints ........ 30

3.4.1.2 Contract Theory ........ 30

3.4.2 Identification and Classification of Exceptions ........ 31

3.4.2.1 General Classifications. ........ 32

3.4.2.2 Classifications According to Causes and Origins ........ 34

3.4.2.3 Classification According to Severity ........ 36

3.4.2.4 Comparison ........ 36

3.4.2.5 Integration ........ 37

3.4.3 Flow of Control ........ 41

3.4.3.1 Exception Propagation ........ 42

3.4.3.2 Responsibility Assignment ........ 43

3.5 Handling Models/Techniques ........ 44

3.5.1 Non-Local Transfer Model ........ 46

3.5.2 Termination Model ........ 46

3.5.3 Resumption Model ........ 47

3.5.4 Retry Model ........ 49

3.6 Summary ........ 52

**4 Object-Oriented Languages and Exception Handling ........ 53**

4.1 Introduction ........ 53

4.2 Why the Object-Oriented Development Paradigm? ........ 54

4.3 Languages ........ 55

4.3.1 Ada ........ 56

4.3.2 C++ ........ 58

4.3.3 Eiffel .......... 60
4.3.4 Java .......... 62
4.3.5 Comparison of Object-Oriented Languages and Exception-Handling Capabilities .......... 64
4.4 Summary .......... 66

**5 Reporting Exceptions .......... 68**
5.1 Introduction .......... 68
5.2 The Necessity of Reporting Exceptions .......... 68
5.3 Information to Include when Reporting Exceptions .......... 69
5.4 Presentation of Information .......... 73
5.4.1 General Issues .......... 74
5.4.2 Natural Language-Centred Issues .......... 76
5.4.3 Human Computer Interface-Centred Issues .......... 77
5.4.3.1 Visualisation Issues .......... 77
5.4.3.2 Structural Issues .......... 78
5.5 Principles to Support Usability .......... 79
5.6 Developments in Reporting Exceptions .......... 81
5.7 Summary .......... 83

**6 Proposals .......... 84**
6.1 Introduction .......... 84
6.2 Identified Problem Areas .......... 84
6.3 Existing Proposals .......... 86
6.3.1 Representation of Exceptions as Objects .......... 86
6.3.2 Adding an Exception Manager .......... 88
6.4 Proposal for a Unified Exception-Handling Framework .......... 92
6.4.1 General Requirements .......... 93
6.4.2 Inclusion of Exception Handling in the Software Development Life Cycle .......... 94
6.4.3 Evaluating the Unified Exception-Handling Framework .......... 96
6.4.3.1 Improvements and Advantages .......... 97
6.4.3.2 Limitations and Disadvantages .......... 97
6.4.3.3 Summary .......... 97

6.4.4 Patterns ........ 98
6.5 Summary ........ 100

**7 Conclusion ........ 101**
7.1 Introduction ........ 101
7.2 Review ........ 102
7.3 Future Research ........ 103

**Appendix A: Exception-Handling Issues in Concurrent and Distributed Systems ........ 105**
A.1 Introduction ........ 105
A.2 Coordinated Atomic Actions (CA Actions) ........ 106
A.2.1 Blocking versus Non-Blocking Communication ........ 108
A.2.2 The Necessity of Concurrent Exception Resolution ........ 108
A.3 Co-operative Object Approach ........ 109
A.4 Summary ........ 112

**References ........ 113**

# AKNOWLEDGEMENTS

I would like to express my gratitude to the following persons for their contributions to the creation and completion of this dissertation:

**My Heavenly Father**

**—For creating humans capable of developing exceptional software, not without exceptions, but also with the ability to make provision for these exceptions.**

**Professor Karen V. Renaud**

**—Who acted as my supervisor,**

and never gave up on me.

For providing me with answers, advice, positive feedback and direction.

**Melanié Malan**

**—From the UNISA Library.**

For help with research material, endlessly updating keywords and many enquiring e-mails as to the progress being made.

**Neil**

**—My husband.**

For consuming so many dinners alone without reproach.

# Abstract


## Summary:

This dissertation investigates current trends concerning exceptions. Exceptions influence the reliability of software systems. In order to develop software systems which are more robust, thus delivering higher availability at a lower development and operating cost, the occurrence of exceptions needs to be reduced and the effects of the exceptions controlled. In order to do this, issues such as detection, identification, classification, propagation, handling, language implementation, software testing and reporting of exceptions must be attended to. Although some of these areas are well researched there are remaining problems. The quest is to establish if a unified exception-handling framework is possible and viable, which can address the issues and problems throughout the software development life cycle, and if so, the requirements for such a framework.



## Keywords:

Exception; Detection; Identification; Classification; Propagation; Exception Handling; Object-oriented; Testing; Reporting; Software Development.

# Glossary

The following terminology is used throughout the dissertation and is defined here for reference purposes and to clarify the meaning of the terms. Different definitions are found in the literature [35, 39, 56] and are condensed to the following:

## Definitions:

- **Fault:**

  An uncontrolled, unintentional or unexpected action taken - a mistake.

- **Error:**

  An error is caused by a fault. An error occasions the malfunction of a software component due to a deviation from what is expected, what is provided for and what can be handled.

- **Failure:**

  A failure of a component is caused by an error. The component fails to function correctly as is required and expected.

- **Exception:**

  An exception is the occurrence of an unexpected event that is not part of the normal processing routine. The occurrence of an exception is due to a fault causing an error that might lead to a failure. It makes normal execution either impossible or undesirable.

- **Exception Detection:**

  An exception is detected by the failure of a validity test and indicates that if processing continues and the value is allowed, it will lead to failure. The system tests values against constraints to ensure that only valid values are accepted.

- **Exception Raising:**

  An exception is raised (within a component) or signalled (between components) after detection and refers to the notification or communication of the occurrence of the exception to the system.

- **Exception Handler:**

  An exception handler is a component developed to address abnormal conditions in order to restore normal operation or controlled termination of the application.

- **Exception Handling:**
  Exception handling refers to the association of a handler with an exception and the execution of the handler.
- **Change in Flow of Control:**
  The flow of control in the component is altered from the normal, passed from the point where the exception occurred and was detected to the location where a handler is provided due to exception raising.
- **Exception Propagation:**
  Exception propagation is the repetitive or continuous change in flow of control. If a handler fails, an appropriate handler can be searched for from a set of handlers in a specified order, the exception thus forwarded to other handlers for possible handling. The process of forwarding continues until a handler is found or a default handler executed.

## Example:

The user is prompted to enter a numerical value. A character is entered unintentionally by *mistake*, a *fault* on the part of the user. An arithmetic operator is applied causing an *error* in the operation. The operation fails to complete execution, since it is impossible, thus causing a *failure*.

If the input is tested for validity, the software *detects* that the value entered is invalid and will cause a failure if it is accepted. A handler is called, *raising* the exception and *changing the flow of control*. If the handler cannot handle the exception, another handler is searched for, the exception thus *propagated*. The *handler* takes corrective action, *handling* the situation, by implementing corrective measures – invalidating the value and requesting the user with the appropriate message to re-supply the input.

# Chapter 1 – Introduction

## 1.1 Motivation for the Research

Exceptions have led to many disasters with results varying from minor to severe. Numerous instances of such disasters have been documented in the literature [32]. Software is central to many life-critical systems and developed by *"error-prone humans and executed on error-intolerant machines"* [53]. All indications are that it will be impossible to develop error-free software in the foreseeable future since software development is done by imperfect humans [53]. The origins of exceptions vary from analysis and design faults to coding, interaction and integration problems. It is not viable to ignore exceptions, since the consequences can lead to the endangerment and loss of human life. On the other hand it is impossible to eliminate the causes, and therefore the symptoms must be treated whenever and wherever they surface. Since prevention is not an option, exception handling is an issue that must be given due consideration.

The indications from the literature are that current exception-handling procedures in software are insufficient – the severe effects that result from exceptions as well as the frequency of occurrences, are ample evidence of this. Maxion and Olszewski argue that *"Exception failures can account for up to two-thirds of system crashes"* [49]. In order to improve on current exception handling a thorough understanding of the nature of exceptions is necessary. Improvements in exception handling will lead to an increase in reliability, availability and ultimately the quality of computer software systems.

## 1.2 Application of the Research

Since it is impossible to develop exception-free software the goal of exception handling is to *handle* exceptions, successfully. The term "successfully" implies reliably. Extensive code verification, in order to eliminate causes, is not feasible in practice [5] due to the vastness of the code itself, the fallible code checking methods and the imperfection of the checkers. As a result, the idea is thus not to eliminate exceptions but to prevent exceptions from causing system failure [53]. The greater the understanding of the nature of exceptions and the problems surrounding them the better the handling mechanisms will become and hence the more reliable the resulting

software. For software to be labelled reliable does not mean that there will be no exceptions, but rather that there will be some form of guarantee about what can be expected in event of an exception.

The following quotations and accompanying comments support the motivation for and the applicability of the research:

- *"Achieving dependable behaviour from complex software systems requires that the software logic must be able to recover from anomalous conditions or exceptional states"* Bail [4].

  -Unanticipated events can occur but reliable systems must be able to function dependably even in such circumstances and provide functional exception-handling mechanisms.
- *"The current lack of effective error-handling techniques for developing dependable object-oriented software produces software components which are usually difficult to understand, to change and to maintain in the presence of faults. Ideally such components should incorporate their exceptional activity in a structured and transparent manner so that the abnormal code would not be amalgamated to the normal code"* Garcia *et al.* [26].

  -Current exception-handling mechanisms are insufficient and there is a need for a separate, well-structured, uniform, implementable, exception-handling mechanism.
- *"With the demand for increasingly reliable software systems and the corresponding increase in their complexity, much attention has been placed on techniques to improve the dependability of these systems"* Bail [4].

  -Systems play an increasingly important role in daily routines and are becoming more complex, and exception-handling mechanisms should escalate correspondingly. As more is expected from computerised systems, more will be expected from exception-handling mechanisms in these systems.

## 1.3 Scope of the Dissertation

This dissertation investigates past and current trends in exception handling throughout the software development life cycle, concentrating on analysis and design issues. It focuses on the aspects of detecting and identifying exceptions, the handling of exceptions and the reporting of

exceptions. It also takes a look at how popular object-oriented programming languages implement the theory and identify problem areas. Aspects of software testing methods used to uncover and prevent possible exceptional behaviour and the reporting of exceptions to the user are also investigated. Problem areas are identified and current proposals to address some of the problem areas are discussed. Although some of these areas are well researched problems still remain. The quest is to establish the feasibility and viability of a unified exception-handling framework, which will address the issues and problems throughout the software development life cycle, and to investigate the requirements for such a framework.

The development of sequential and centralised software systems is a special, simplified case of the development of distributed and concurrent software systems. The issues that are dealt with in the body of this dissertation are general and valid for centralised as well as distributed and concurrent systems. Specific additional details concerning distributed and concurrent exception handling are described in Appendix A.

Generally most of these issues are dealt with superficially in order to capture the essence of exception handling and the related inadequately addressed problems. Since each of these issues could easily form the subject of a separate dissertation, details to support the arguments and conclusions are limited.

## 1.4 Structure of the Dissertation

The rest of the dissertation is structured as follows:

- Chapter 2 contains information about the causes and origins of exceptions and the role testing methods play in uncovering exceptions and measuring the success of the provided exception-handling mechanisms.
- Chapter 3 contains general background regarding exceptions in the software development life cycle, general policies on exception handling and the strategies for implementing the policies. It deals with the different aspects concerning the exception life cycle, from detection to handling.

- Chapter 4 investigates and compares how the current theories described in Chapter 3 are implemented in object-oriented programming languages such as Ada, C++, Eiffel and Java.
- Chapter 5 explores the issues surrounding the reporting of exceptions to the user, such as the information to be included in the messages as well as the presentation of the information, and it motivates the necessity of reporting exceptions.
- Chapter 6 summarises the problem areas of the preceding chapters and discusses proposed solutions to address some of the issues mentioned. The quest is to establish if a unified exception-handling framework is possible and viable, which could address the issues and problems throughout the software development life cycle, and if so, the requirements for such a framework.
- Chapter 7 concludes with a brief review of the contents.
- Appendix A introduces concepts developed especially for concurrent and distributed systems as they are more complex than centralised systems and require additional constructs in order manage these complexities.

# Chapter 2 – Being Realistic About Exceptions

## 2.1 Introduction

The exception-handling constructs in object-oriented languages in common use seem at first to be fairly straightforward and sufficient. They would appear to be easy to implement during the coding phase since provision can be made for expected as well as unexpected exceptions. Commonly occurring exceptions are even provided for by means of built-in handlers and the programmer can define handlers for any other exceptions. It therefore seems to be unnecessary to spend time on issues regarding exception handling during the other phases of the software development life cycle.

However, such assumptions are obviously misleading. Fatal exceptions, programs being "hung" and system "crashes" occurring in operational software would indeed be rare events if this were true. Reality - in practice as well as in the literature - indicates the contrary. Evidence of this is found in the fact that programs still experience situations that are not handled effectively by existing exception-handling constructs, allowing normal processing to continue despite the occurrences of exceptions with serious consequences. Many users, as well as developers, experience this and it is well documented throughout the literature.

This chapter will explore the precautions taken to avoid and eliminate exceptions as well as the persistence of exception occurrences despite such precautionary measures. Section 2.2 will shed light on causes and origins of exceptions and the reason(s) why it is not possible to eliminate all exceptions and the underlying problems that make exception handling not as straightforward or as easy to deal with as first impressions may indicate. Many software developers' sole defence against the vagaries of exceptions is thorough testing. They hope that, by means of testing, they can eliminate or at least appreciably negate the effects of exceptions. Since testing forms their sole defence, Sections 2.3, 2.4 and 2.5 will look in some detail at current practices in testing software. Section 2.3 introduces general background on software testing. Section 2.4 discusses different testing techniques while Section 2.5 categorises the testing results in terms of the effects they have on the system and possibly on the environment. Section 2.6 follows with a summary of the findings of the Chapter.

## 2.2 The Causes and Origins of Exceptions

As mentioned in Chapter 1, software is developed and operated by error-prone humans. Exceptions are thus raised in the system to safeguard the system from failures that can be caused, by amongst others, errors originating from the user. The two main sources of exceptions are code defects and erroneous data entered by the end-users [4].

Code defects are introduced throughout the entire software development life cycle. Code defects may be due to contradictions, errors or misinterpretations in the system's requirements and specifications [8, 32, 33]. This in turn may be due to inherent complexity, miscommunication or changes in the environment with respect to the specifications [4, 32, 46, 53]. Developers and programmers are imperfect and make mistakes so that design and programming errors are also causes of exceptions. The pressure of unrealistic schedules and deadlines can further increase the likelihood of errors made due to guesswork [32]. Even erroneous compiler code can lead to defects. Inadequate testing can also allow some errors to remain in the system. Furthermore, incomplete documentation used as a basis to modify and maintain code, can also introduce errors due to incorrect assumptions being made. Development tools can also be the source of errors as they are developed by imperfect developers and are subject to the same problems as any other software product [32]. Other possible sources of exceptions include physical component failures. Code defects can be lessened by testing, but not eliminated.

Erroneous data entered by the user include data entered in the wrong format and of the wrong content. These are more obviously expected and provided for with exception-handling mechanisms but can never be eliminated since it is impossible for a user *never* to err.

Kuhn [41] describes the causes of exceptions as due to code defects and/or the entering of erroneous data, as being either incorrect or missing constraints. This corresponds with the categorisation given by Maxion and Olszewski [49], of exceptions originating from system developers when designing a new system. Their two main categories are exceptions occurring due to errors of commission and errors of omission. Their categorisation is compiled as follows:

- Exceptions due to errors of commission:
    - Selection exceptions
    - Sequence exceptions
    - Timing exceptions
    - Qualitative exceptions

- Exceptions due to errors of omission:
    - Recall exceptions
    - Unknown exceptions

In the first category, where making the wrong decisions in existing constructs cause exceptions, it is possible to find and eliminate some of the exceptions through inspection and related techniques. Wrong decisions may include those made regarding selections (wrong choice), sequences (wrong order), and timing (too early or too late) or quantity-related (too little or too much) [49].

In the second category, where the absence of constructs causes exceptions, it is very difficult to find the origin of the problem since there is often no evidence to indicate that something is amiss and there may be no evidence to provide a clue or initiate recall or memory. There are two types of exceptions of omission – omitting *known* and *unknown* exceptions. Recall exceptions are caused by "forgetting", which can be either partial or total, exceptions *known* to occur which should have been thought of, but were forgotten about. In this case a list of common exceptions caused in this manner can be compiled and used as a checklist to aid the recall process. *Unknown* exceptions are the most difficult exceptions to discover but may also be the most dangerous not to uncover. These include new exceptions – exceptions arising for the first time ever and exceptions not thought to be applicable in the specific situation. These may only surface during actual operation and may be very expensive to correct.

Strong and Miller [76] found that to find all possible exceptions in code a 100% manual inspection is necessary, which is not always possible due to time constraints and the vastness of the code not to mention the imperfection of the checkers be they automated or human. Even then, it might not be enough since the information needed for judgement, to decide if a certain condition may lead to an exception or not, may not be readily available. They also found that the programming control structures are too restrictive in relation to the requirements. Real life situations must often be modelled in terms of mathematical constructs not always allowing a one-to-one mapping. The variety, possibilities and the inconsistencies found in reality may render such a mapping impossible or such a mapping may be too difficult and time consuming to develop. The limitations may be due to the fact that the more restrictive a language is, the easier it is to develop and guarantee the correct functioning of the language. The authors also deduct that the exception-handling mechanisms do not reflect the dynamics of a changing organisation

and that not all the exceptions are software or logic exceptions. They suggest that systems should not be designed to eliminate humans. It should be kept in mind that humans are better at certain tasks than computers and that systems should be designed with co-operation between the user and the system in mind to eliminate exceptions. Also, some exceptions might be related to physical-operation exceptions such as where a system is not configured or operated correctly.

To summarise - exceptions can thus originate either from the developers or from the end-users of the software. Exceptions are "built in" through the whole of the software development life cycle by either making incorrect decisions or by the omission of constructs. Although it may seem that all the blame can be laid at the doors of the developers and end-users this is not the case. Due to physical component (hardware) failures, changing environments, platforms and software interaction profiles, some of the exceptions that occur are beyond the control of the developer, but even so the responsibility of anticipating such exceptions lies with the developer.

The notion from Chapter 1, that exceptions are a reality that cannot be wished away, is thus once more reinforced. Testing must be conducted prior to the release of the software to minimise the defects remaining in the code and also to establish the effectiveness of the handling mechanisms prepared for those exceptions that cannot be eliminated. The procedures and tests for the efficiency of exception handlers and for possibly uncovering unpredicted exceptions are not different from the existing software testing. They involve a shift of focus specifically aimed at the uncovering of any flaws in the exception-handling mechanisms and the software itself with regard to exceptions. It may seem that thorough and complete testing can easily solve the problem of uncovering exceptions and testing the efficiency of exception-handling mechanisms. Section 2.3 takes a closer look at some of the existing testing techniques and their applicability to the problem of uncovering unexpected exceptions and erroneous handling mechanisms.

## 2.3 Software Testing

Software testing can be described as the evaluation of the results of the operation of a system or application under controlled conditions [32]. These conditions include normal or expected conditions as well as abnormal or unexpected conditions. The goal of testing is to find and prepare for exceptions before they surface in software operation [62] in order to improve the reliability and the quality of the software [3]. Testing is conducted to eliminate, where possible, the causes of exceptions but also to test the existing exception-handling constructs to establish

whether they provide adequate coverage of exceptional conditions [49]. The earlier testing is conducted the sooner problems will be identified and addressed leading to a reduction of cost and development time. The main focus of the testing techniques and issues discussed here is on the influence they might have on discovering and handling exceptions, rather than that of verification ("Are we building the product right?") and validation ("Are we building the right product?") [62].

Testing techniques test the level of robustness of a system where robustness can be defined as [40]:

- *"... the time between operating system crashes under some usage profile."*
- *"the degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions."*

A systematic and structured approach is necessary to ensure that testing is conducted thoroughly and completely. Unfortunately, as Maxion and Olszewski find *"In operational computer software systems often more than two thirds of the code is devoted to detecting and handling exceptions. Yet, since exceptions are expected to occur rarely, the exception handling code of a system is in general the least documented, tested and understood part"* [49] which makes testing all the more difficult.

The following is necessary when conducting software-testing [33]:

- *The requirements of the software module undergoing testing.*
  In order to establish if a test was successful, the results of the test have to be compared with some norm of what the results should be. The requirements give an indication of what the desired behaviour of the module should be, and are used to evaluate the results.
- *A test plan.*
  The objectives, scope, approach and the focus of the testing procedure is described in the test plan document. It is necessary to define exactly what is being tested in order to establish if the results indicate success or failure.
- *Test cases.*
  A test case aims to find an undiscovered error [62]. Test cases are documents containing details of the input used for the test, the action or event on the input and the anticipated response.

- *Reviews of test cases.*

  During a review a comparison is made between the results obtained from the test and the anticipated results. If the results do not match, the problem must be identified, communicated to the responsible parties for resolution and then re-tested.

As mentioned earlier in this section software testing must preferably be conducted from the earliest phases of the software development life cycle. Since it seems that some resilient errors always remain in the software, it follows that testing should never cease [62], which means that software will never reach the stage of being fully operational. The question arises now as to when testing can cease. The following criteria can be used to decide when to end testing [33]:

- Testing will have to cease when a deadline has to be met.
- The test budget is another factor that can be used as a criterion. If it has been fully spent it is no longer possible to continue testing.
- When a number of test cases have been completed and a certain percentage of them have passed the test, an acceptable degree of reliability can be guaranteed.
- If the code, functionality or requirements have been covered by testing to a certain level.
- The number of errors uncovered falls below a certain level.

The last three criteria should be used in preference to the first two since the first two do not take into account the extent to which testing has been conducted as is done by the latter, but in some cases it might be unavoidable to apply the former. A test can be regarded as successful if it uncovers an exception that exists and of which the developers were unaware [62].

A software system can be tested in many ways during the different phases of the software development life cycle. First of all the system can be tested as a whole as if in full operation [83]. This is the most obvious type of testing – to test the user interface and the erroneous types of input the system will have to be able to handle. Although it is necessary to test large systems as a single whole entity, it is very difficult to do. The results obtained from such a test will tend to be ambiguous due to the large number of interactions that might or might not have influenced the results. To isolate the location of the origin of an exception, module testing must be performed [32, 62, 83]. Programmers have a normal tendency not to be able to be objective towards their own work. It is therefore important to also have other people perform inspections and analyses on the system. Different people think and reason differently and this could lead to the identification of unforeseen causes of exceptions. Automated tests may be performed but they

involve a serious overhead in terms of space utilisation, execution time and complexity. It may be a faster method of testing but not necessarily cost effective and since these automated tests are also developed in the same way as other software, they might themselves be subject to exceptional behaviour.

System testing can be performed with the aid of many different techniques during the different phases of the software development life cycle. Some of the different techniques or methodologies will be listed in Section 2.4 and discussed, with specific reference to exception handling.

## 2.4 Testing Techniques

The different testing techniques include Collaboration Techniques, Execution-Style Techniques, Multi-version Techniques and Mathematical Techniques. These techniques will be discussed in more detail in the subsections that follow in order to establish their worth in the process of uncovering overseen exceptions and determining the efficiency of the provided exception-handling mechanisms.

### 2.4.1 Collaboration Techniques

Collaboration techniques are characterised by developers working together, co-operatively. It works on the premise that two (or more) heads are better than one and that different people work from different perspectives, and the resulting system might emerge with fewer exceptions. This testing technique investigates the existing system and it is more likely that contradictions between what is expected and what is found will be discovered, rather than uncovering what is not there. Also, the developers of a system will usually be involved in the testing process and their close involvement with the system does not foster objectivity, which might also lead to oversights and hence to exceptions. Collaboration techniques are thus good for discovering exceptions due to errors of commission, but not of omission [49].

#### 2.4.1.1 Dependability Cases [49]

The aim of this method is to systematically identify the potential risks (i.e. exceptions) in a system. Essentially it is a framework for creating checklists to consider exceptions and the

conditions under which they occur and thus give structure to the manner in which programmers think about exceptions. These checklists serve the purpose of improving recall of possible exception conditions. An example of a well-known mnemonic used for such a checklist is the "*Everyone knows exceptional children*" [49], where each of the letters of the word *children* reminds of a common type of problem to be verified.

Exceptions are classified into two groups – those to be tolerated and those to be avoided. In order to identify and list the risks of a system, hazard analysis is done. This is done through the use of Fishbone diagrams. (Also known as Cause-and-Effect - or Ishikawa diagrams) [49]. The diagram looks like a fish skeleton where the head depicts the phenomenon to be avoided, the "ribs" the categories of events that cause failures and between ribs examples of specific causes. The ribs are named according to the *children*-mnemonic. An example of such a diagram can be seen in Figure 2.1:

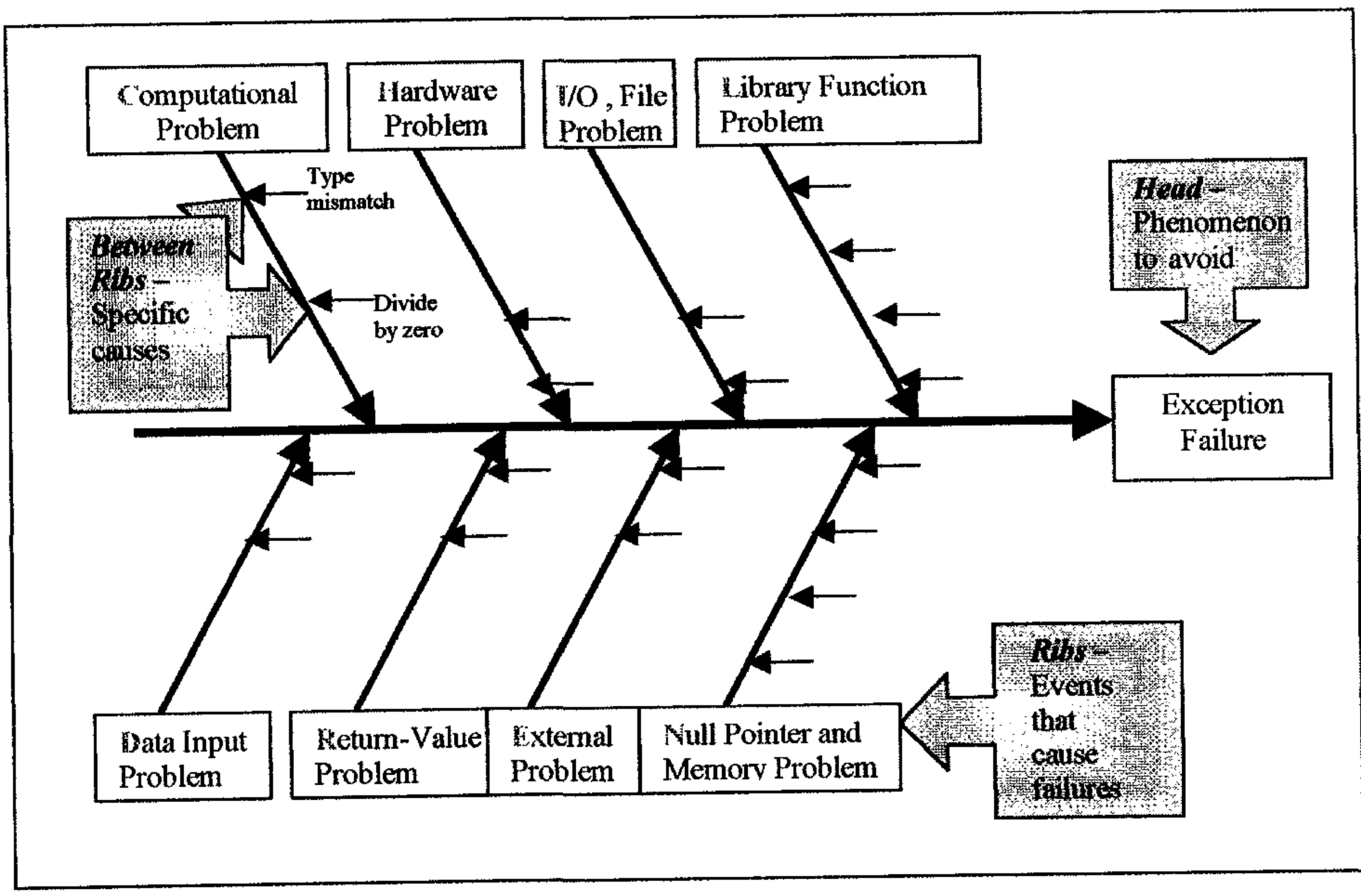



**Figure 2.1 – A Fishbone diagram [49].**

Two types of test cases are required to completely validate the code under test:

- Normal test cases
- Fault test cases

Normal test cases are used for what a system should do and fault test cases for what it should not do, in other words it address the exception conditions and the provision made for them in code. *"There is a great demand for a universal, one-size-fits-all checklist"* [49]. Since there are many exception conditions and it is difficult to remember all of these conditions, a checklist may prove useful, but it may also give a false sense of security since the programmer may feel that if each item in the list was checked there would be no exceptions left.

#### 2.4.1.2 Reviews [55, 62]

A review is a technique for debugging and improving a system in order to deliver a product with a higher degree of predictability and quality [55, 62]. Reviews attempt to confirm the correctness of what is already done and to point out possible improvements. Developers usually perform reviews but end-users can also participate. Due to difficulties and costs involved, the extent to which users aid in the testing is limited [55]. It serves the purpose of uncovering defects and should be conducted at various points through the life cycle. There are many types of reviews. Two types will be discussed briefly here - Inspections and Walkthroughs [62].

Inspections provide feedback to the developers on the usability and quality of their product. By pointing out problems, they therefore serve to improve the quality [6]. During inspections, evaluators concentrate on the interface, since it will form the point of interaction between the end-user and the system, and the user perceives the whole system through the interface [55]. Commonly used forms of inspections are formal usability, feature, consistency and standard inspections. Formal usability uses a formal procedure, feature inspections concentrate on specific features, and consistency inspections test if the design is consistent with the requirements and the real-life situation and standard inspections are done to measure compliance with certain set standards [55].

Walkthroughs evaluate the features of the interface systematically [44]. Different types of walkthroughs exist, namely, cognitive and pluralistic (co-operative) walkthroughs. Cognitive walkthroughs investigate the logical order of the functionality provided by the system while pluralistic walkthroughs are done by developers in co-operation with end-users [55]. It simulates the user's problem solving process and oversees that the dialogue is supportive of the goals to be achieved by and the memory content of the user [55]. A walkthrough generates feedback on the design and can reveal shortcomings in the understanding, and as such, the representation of the problem domain.

### 2.4.2 Execution-Style Techniques

Execution-style techniques are implemented by executing the software system and subjecting it to different conditions to evaluate the system's reaction. It typically tests the system as a whole, but can also be used to test components, as if implemented and fully functional by means of actual execution. Developers are mostly involved with this testing technique. Again, as with collaboration techniques, it tests the existing system and does not impel the testers to test for missing constructs. It is therefore also a good technique to test for exceptions due to errors of commission but not of omission.

#### 2.4.2.1 Balista Robustness Testing [40]

This robustness testing technique is used to determine how well a software module handles exceptions and is not concerned with functional correctness. It uses combination tests of valid and invalid parameter values, also varying the ordering of the values, for system calls and functions. A module is then executed with each set of parameters. This technique can be useful to establish whether the input from a user will be accepted or not.

#### 2.4.2.2 Black Box / Functional Testing [40, 81]

This technique requires a model of the functionality of the system undergoing testing. It attempts to ensure that the system meets its specifications from a behavioural or functional perspective [74]. Testing is conducted at the level of the interface to determine if it meets the operational functionality as set out by the requirements [32, 62]. It evaluates the output delivered from specific input provided, with little regard as to how the output is derived from the input. Although it is more successful in evaluating the existing functionality it could also be productive in indicating missing interactions.

#### 2.4.2.3 White Box Testing [62]

This technique tests the system at code level. The functionality of each independent path in conditional structures is tested, the logic of the conditions, the boundaries of repetitive structures and internal data structures are also tested for validity [32, 62]. Due to combinatorial problems it is impossible to conduct exhaustive testing of all the possible paths. This technique may be useful to detect possible exceptions related to branching, logic, and boundary constraints.

### 2.4.3 Multi-Version Techniques

These techniques make use of redundancy. More than one copy is created independently to investigate the differences, and hopefully, through the investigation exceptions could be discovered in the different versions. These techniques are superior for discovering exceptions due to omissions. The probability that the same exception will be omitted in all the versions by all the developers is small. These techniques are helpful in distinguishing between non-exceptional and silent exceptions. If no exception is reported is does not necessarily indicate that there is no exception - it might be a silent exception - an exception that slips through without any indication of its occurrence.

#### 2.4.3.1 N-Version Programming [47, 49, 53]

Three or more - n-versions - of the system or module are independently designed from the same specification. Although it can be used for module-level testing, it is more often used for system-level testing [53]. The versions are then tested with the same input, with read-only access to the input, to ensure that the input remains identical for all the versions and is not changed by any version's execution. This technique is used to distinguish between non-exceptional conditions and silent exceptions. Various versions will disagree as to the results for identical tests [40] and investigation of the disagreements can lead to the uncovering of exceptional conditions.

A combination of erroneous and non-erroneous parameters, tested in different sequences, may result in non-exceptional conditions. This may be due to the order in which the values are tested and accepted. Some values may be accepted regardless of others. An adjudicator determines the correct or best output, based on majority-based voting, confidence voting and median voting. The success of this technique depends on the voting check to identify the erroneous output of a faulty version.

#### 2.4.3.2 Recovery Blocks [53]

Recovery blocks aim to provide fault-tolerant functional components that may be nested within a program. This technique is used mostly for module-level testing. A number of modules of different design from the same specification are created. The preferred design is named the primary module and the rest the alternate modules. On entry to a recovery block, a recovery point is established which allows the program to restore to this state, if required. Acceptance tests check the state of the program for successful operation. If a failure occurs, the module is restored

to the recovery point, and an alternate executed. This process continues until an alternate executes successfully and is accepted or until all the alternates fail. Acceptance tests are implemented to detect the occurrence of exceptional conditions and can make use of a number of approaches to establish if the execution was successful.

### 2.4.4 Mathematical Techniques

Mathematical techniques are the most rigorous and make use of formal techniques to prove correctness. They deliver the most certain results. They are, however, very complex, time consuming and require more mental effort than any of the other techniques [3, 62]. This technique is applied to critical components rather than whole systems. Mathematical proofs can also not be accepted without human verification [42]. Not all developers will be able to engage in this form of testing due to the mathematical background requirements for these techniques.

#### 2.4.4.1 Correctness Proofs

In order for a correctness proof to be carried out there are prerequisites that must be met. Correctness proofs require that the specifications of a component must be correct and precise and must formulate concepts formally but still be humanly understandable. It is a very complicated and difficult process to set up the specifications according to these requirements [42]. A correct method or solution is also required to solve the problem and then, lastly, the correct method or solution, must be implemented correctly [55]. If any of the prerequisites are not met, a correctness proof cannot be conducted. From these it is proven that if the requirements are correct, the proposed solution is correct and if implemented correctly, then the component will work correctly without the need for further testing. If this technique is applied to certain modules the cost of testing can be reduced as the need for testing certain modules can be eliminated. This can also reduce the development time [3].

### 2.4.5 Summary of the Testing Techniques

These testing techniques are not perfect – if they were, software would be exception-free. But they are a definite aid in uncovering exceptions that might otherwise remain in the software, as well as inadequate exception-handling mechanisms that might have serious effects on the system. Software testing increases the development time and also the cost of development, making the final product more expensive but more reliable - and so worthwhile. In Section 2.4.5 a summary

is provided of these techniques and Table 2.4 tabulates not only the most important characteristics but also the advantages and disadvantages of each methodology in general.

| Technique: | Collaboration Techniques | Execution -Style Techniques | Multi-Version Techniques | Mathematical Techniques |
|---|---|---|---|---|
| Section Reference: | 2.4.1 | 2.4.2 | 2.4.3 | 2.4.4 |
| Examples of Specific Testing Techniques: | Dependability Cases<br>Inspections | Balista<br>Black Box Testing<br>White Box Testing | N-Version Programming<br>Recovery Blocks | Correctness Proofs |
| Description: | Developers working together, in co-operation thinking and talking about the system under development. | Executing the system or parts of the systems and subjecting it to different conditions. | Alternatives for components are created independently and their differences investigated. | Makes use of formal techniques to proof correctness under conditions. |
| What is required for the test to be conducted: | Requirements specification. | Interface (for Balista & Black Box)<br>Code (for White Box) | Alternatives in code. | Requirements, proposed solution and the implementation of the solution. |
| To uncover exceptions / To test efficiency of exception-handling mechanisms: | Exceptions | Both | Both | Exception-handling is functioning correctly |
| Who tests: | Developers and end-users | Developers and end-users | Developers | Developers |
| Advantages: | Discovering of exceptions due to commission. | Discovering of exceptions due to commission. | Distinguishing non-exceptional from silent. | Results can be trusted.<br>Reduces testing. |
| Disadvantages: | Exceptions due to omission might go undetected. | Exceptions due to omission might go undetected. | Problems are experienced with the creation of acceptance tests and voting checks in order to find the "best" alternative. | Time consuming.<br>Complex.<br>Skilled developers needed. |

**Table 2.2 – Summary of the different testing techniques.**

These techniques test different aspects of exceptions. Collaboration techniques try to uncover exceptions due to errors of commission by conceptually working through the system on a high level from the requirements and design specifications. Execution-style techniques aim at the same goal as collaboration techniques but they use an operational approach whereby an actual component is tested, thus they test at a more detailed level. Multi-version techniques aim to uncover exceptions due to errors of omission in a component through creating multiple versions and comparing them, whilst mathematical techniques are employed to prove the correctness of a component and eliminate the need for further testing of the component.

Not one of these techniques can be used to test all the aspects of a system for remaining exceptions. Collaboration techniques can be used during the earlier development phases where operational modules are still non-existent. Execution-style techniques can be used during later development phases when the operational modules become available. Multi-version and Mathematical techniques can be used for more specific purposes in addition to some other testing techniques. They require more effort than the previously mentioned techniques but might be worth the additional effort for critical components.

It would be useful if an overall exception-handling policy could be established that follows through the different phases of the development life cycle. Appropriate testing techniques can be applied during the different phases to test for specific types of exceptions. This would benefit the development process as with each phase more assurance could be given as to what was guaranteed. Fewer modifications would have to be made during the testing phase as errors would be discovered earlier in the development process and corrected before spreading and infecting more of the system.

The next section investigates the effects of exceptions on the user, the system and the consequences for the environment.

## 2.5 Categorising Results

The results from the different testing techniques can be categorised with the use of the CRASH severity scale [40]. The scale is used for exceptions not successfully handled and uses each of the letters of the word "CRASH" to represent different levels of severity of the effects of an occurring exception if it should occur in an operational system. The level in the "CRASH" scale is an indication of the seriousness of the effect of the exception and can be used to prioritise the exception-handling process. The priorities can be assigned as illustrated in Table 4.3.

By categorising and prioritising the results it might be possible to lessen the effect of the exception on the user, the system and the environment. Categorisation and prioritisation are an aid to structured thinking about exceptions and the possible effects of the exceptions.

| C | R | A | S | H |
|---|---|---|---|---|
| ***Catastrophic*** | ***Restart*** | ***Abort*** | ***Silent*** | ***Hindering*** |
| In this case the operating system is corrupted and the machine "crashes" and reboots. | An applications task becomes "hung" and intervention is necessary to terminate and restart the current task. | Indicates the abnormal termination of a single task. | No indication of an exception is given. It is very difficult to distinguish this type of exception. (If no indication of an exception is given it does not necessarily mean that there is no exception.) | An exception occurred but is incorrectly classified through, for example, an inaccurate exception code, thus misclassified. |
| | | **LESS SEVERE** → **LOWER PRIORITY** | | |
| ***Resolve Immediately*** | ***Give High Attention*** | ***Normal Queue*** | ***Low Priority*** | ***Defer*** |
| This indicates that the exception is extremely serious and that further development depends on the resolution of this exception and cannot continue unless the exception is resolved. | Indicating that it is urgent that the exception must be resolved and that further development must be continued cautiously until the exception is resolved. | The exception will be dealt with in time and it is not urgent. It will be attended to in the normal course of development. | The exception must be resolved but it can be left for a later stage. It is not influencing the rest of the development process in the current stage. | Resolving the exception can either be ignored or put off until the development of a new version. |

**Table 2.3 – Severities and priorities [16].**

The severity of an exception not handled successfully can have different implications for the user, the system and the environment. The effects and consequences of the exception will be more severe and expensive in correspondence to the dependency of the user and the environment on the system. Possible effects are listed and discussed below:

- For the user directly involved, it can vary from inconvenience to irritation and even frustration [3]. This will pass, except for the unpleasant memories, but it may influence the user to choose a different product in future but otherwise it will have no lasting effects.

- On the system, it can have a variety of effects ranging from none to total malfunction. It can also mean that the system can be offline for anything from a few minutes to a few days. More seriously it can result in impaired functionality and inconsistent resources. Systems can be repaired, even by replacement, and the exception will have no permanent effect as the functionality can be restored.
- The most serious of effects, and in most cases irreversible, will be on the environment. Firstly, for users in the environment it might imply endangered lives or even result in the loss of lives [3] and secondly, for processes dependent on the functionality it can imply processes unable to complete or proceeding erroneously. For example, in the case of a production cycle controlled by a computerised system it might mean that the production process must be halted or that products are manufactured with defects leading in both cases to financial losses not easily reversible. There may be an irreversible waste of money, time and effort [3].

## 2.6 Summary

Despite the variety of different, sophisticated and high standards of testing techniques developed to test different aspects of software during the various phases of development, exceptions still frequently occur in operational software. Testing plays an important role in minimising the exceptions remaining in software but it cannot guarantee the elimination of *all* exceptions. It is thus imperative that a complete study of exceptions must be made in order to be able to develop improved mechanisms and strategies for exception handling through a better understanding of the intricacies of the underlying problems surrounding exceptions and exception handling.

The information provided in the preceding sections confirms the initial indications that exceptions are a reality that must be dealt with. It was confirmed by investigating the causes and the origins of exceptions in Section 2.2. This section showed that some exceptions could be eliminated but, more importantly, that there are others that cannot be eliminated and that there is a need to test software for overlooked exceptions and the functionality of exception-handling constructs. Section 2.3 introduced software testing in general and Section 2.4 discussed testing techniques and compared these. Section 2.5 classified test results since not all exceptions have equal effects on a system. The next chapter, Chapter 3, reports on issues surrounding the life cycle of an exception, from detection to handling.

# Chapter 3 – Background

## 3.1 Introduction

The goal of any development process, including software development, is to ensure the highest quality product possible at the lowest possible developing and operating cost [49, 59]. To realise this goal the occurrence of exceptions needs to be reduced and the effects of such exceptions controlled. The occurrence of exceptions is evidence of an imperfect product and directly influences the perceived reliability, and, as such, the quality of the software system [4, 80]. The number of exceptions that arise during a software product's execution is *not* an indication of its quality, since the execution depends on factors beyond the control of the software developer, such as, for example, the environment at execution time. The reaction of the product to exceptions *is*, on the other hand, a definite measure of the quality.

There are three basic perspectives with respect to the handling of exceptions, namely: [76]

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

Not all exceptions are unpredictable and neither are they random events. The effects and the possible consequences of exceptions in systems on which human lives may depend make it impossible to ignore such events given the possible severity of exception effects. The first approach is thus rendered invalid. The second approach is unrealistic. Exceptions can be caused by errors, but even in the case of error it is not always possible or cost-effective to find and eliminate the cause. It is impossible to predict and eliminate all possible exceptions during the systems-implementation process, especially when the software is running on different platforms, different operating systems and interacting with a variety of application software in a dynamic environment.

The third view reflects reality and is the only feasible option. It is clear that exceptions are a reality that must be dealt with because of the possible severity of exception effects and unpredictable frequency of occurrences [49, 76, 83]. Exceptions are a real threat to any system, not only for the present, but also for the future. Software will never be exception-free, thus the better the handling mechanisms developed, the more the quality improves. The

efficacy of exception-handling mechanisms influences the reliability of all software systems and so influences the experience of all users interacting with a computerised system in one way or another.

The approach in this dissertation will be that an exception is a real threat to normal program execution that should be dealt with. In the literature a wide variety of definitions can be found for exactly what constitutes an exception:

- *"An exception is a special condition, usually an error, that interrupts the normal flow of program execution"* [57].
- *"Exceptions in our view refer to facts or situations that are not modeled by the information systems, or deviations between what we plan and what actually happens"* [46].
- *"Exceptions, situations that cannot be correctly processed by computer systems, occur frequently in computer-based information processes"* [76].

There is still a question of *how* they should be dealt with, and *at what level* of the system development process.

In order to handle exceptions effectively a thorough understanding of exceptions in the software life cycle is required. This includes:

- Uncovering of possible causes and origins.
- Detection.
- Identification.
- Classification.
- Handling.
- Implementation.
- Reporting.
- Testing.
- Effects on the user and the system.
- Consequences on the environment.

The different aspects that will be addressed in this chapter are listed below and depicted in Diagram 3.1. Current issues regarding exception handling, including the influence on the different software development life cycle phases will be addressed in Section 3.2, general policies regarding exception handling in Section 3.3 and the strategies implemented in realising the policies and different handling methods will be discussed in Sections 3.4 and 3.5. Other issues are dealt with in the chapters that follow. Concurrent and distributed systems

have additional issues that must be taken into account due to the inherent complexity of these systems. Issues specific to these systems are mentioned separately in Appendix A.

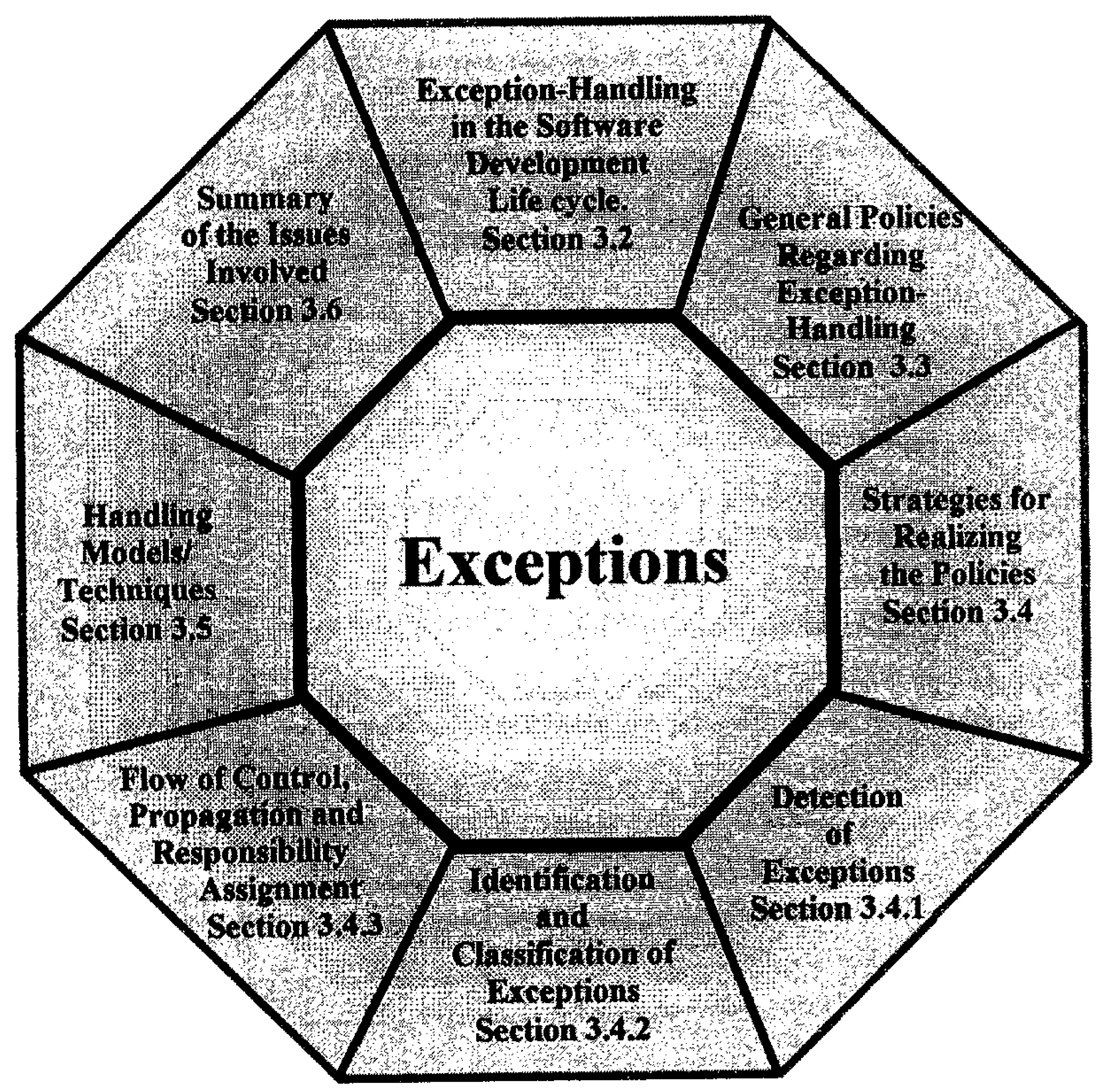



**Diagram 3.1 – The issues related to exceptions and the sections discussing the issues.**

## 3.2 Exception handling in the Phases of the Software Development Life Cycle.

The software development life cycle consists of the phases of requirements determination, analysis, design, coding, testing, implementation and maintenance. Many authors support the notion that exception handling should form an integrated part of the entire software development life cycle, from the earliest phases onwards [17, 34, 52, 58, 61, 75], whilst others are of the opinion that it should be supported from the design phase onwards [4, 10].

The later the exception-handling issues are attended to the greater the loss of context of the exception with regard to the rest of the system and throughout the development process [17,

86]. Owens [58] proposes that a general policy be set out on exception handling for the whole software development project and so does Stewart *et al.* [75] as indicated by their statement "In every software, project architects and designers face the task of defining strategy and tactics for handling exceptions in the application under development". An overall policy will influence tractability early on and throughout the project and between phases to ensure that the requirements concerning exceptions are adhered to in each phase [4, 34]. The context and correlation between exception states can be lost if not incorporated from the beginning [17]. If this is done, exception handling will also be able to enjoy the benefits of iterative refinement during the software development process, starting at an abstract level and working towards a more detailed level [4]. To allow for maximum flexibility the policy should be language independent [30, 58]. This approach enhances maintainability, since, when developers return to the project for enhancements later, the policy will be well documented and so too the strategies used to implement the policy. It will also ensure that exceptions are anticipated and dealt with uniformly throughout [4, 9, 46, 58, 71].

A possible reason for addressing exception-handling issues only in the later phases of the life cycle is the possibility of adding complexity to the already complex development process. From the literature the increased complexity resulting from added exception handling seems to be a minor factor in the light of later problems of a seemingly greater magnitude, such as, for example, tractability, which results from ignoring exception handling during systems-development.

Exception handling is currently dealt with almost exclusively during the coding phases of development. This leads to inconsistencies, because it is added without considering the structure of the rest of the system [4]. If the design does not make explicit provision for exception handling there is no choice but to add it in an *ad hoc* fashion. Such uncontrolled and unsystematic provision for exception handling can do more harm than good [8]. Naturally different aspects of exception handling must be attended to during the different phases of the software development life cycle so as to complement the aim of each phase. This will be discussed in more detail in Chapter 6.

Although it is impossible to guarantee exception-free software, this does not have to mean that there can be no certainty about what can be expected. To a certain extent guarantees can be given as to the predictability of system behaviour with respect to exception handling. Austern [2] suggests defining safety levels with each level containing a higher degree of safety than the previous one by adding some feature to those available at the previous level.

For example, level zero can be defined as giving no guarantee and level four as being exception-safe with a full commit and rollback capability as depicted in Figure 3.2.

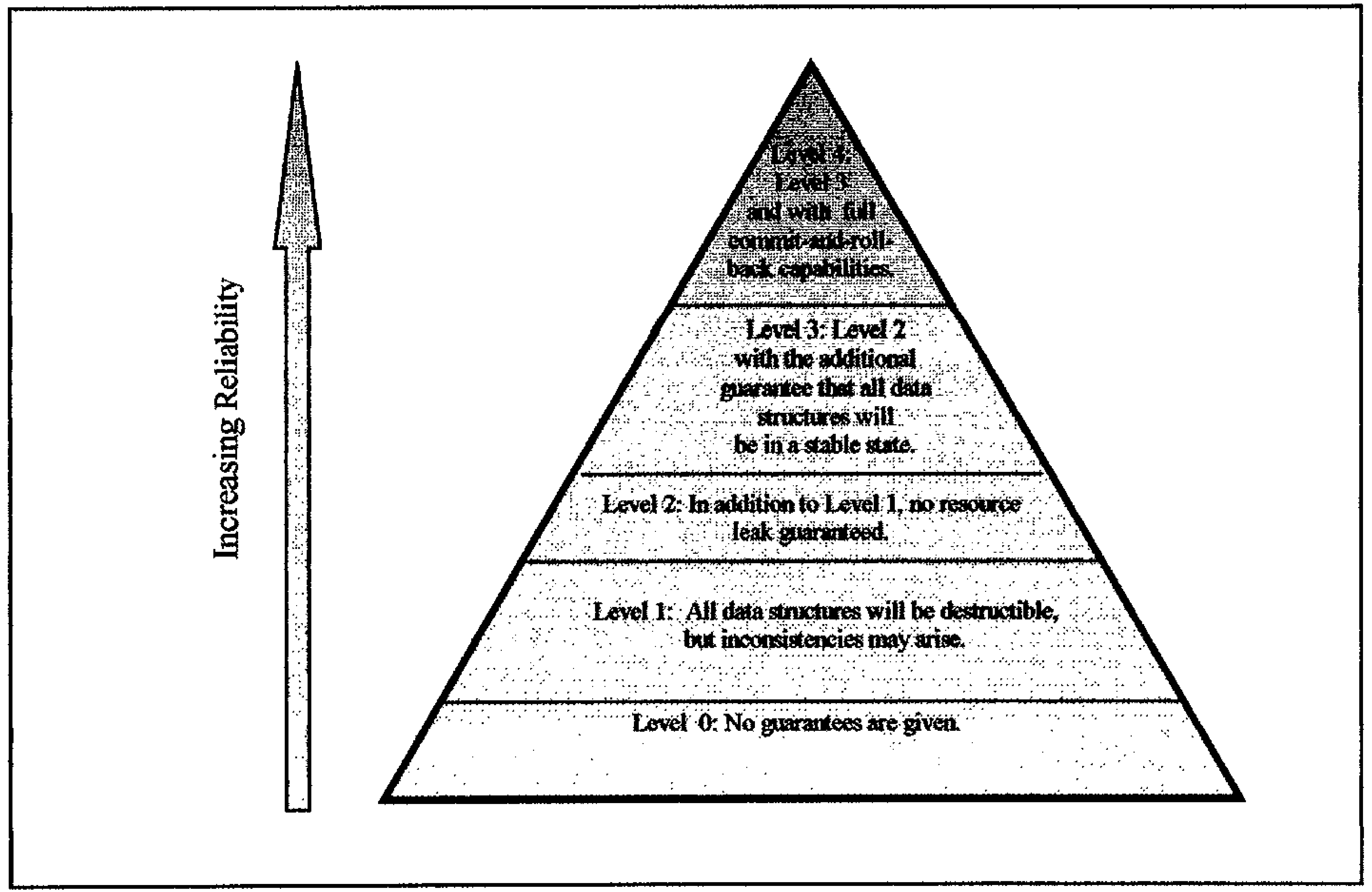



**Figure 3.2 – Safety levels and reliability [2].**

Exception safety indicates that software is functional in the presence of exceptions [77]. The rest of the levels then guarantee predictability in between the two extremes [2]. The main idea is that the lowest level indicates no guarantee while the highest indicates reliability even if an exception occurs.

Existing policies for dealing with exceptions are summarised in the following sections. These policies can be expanded and form the basis, at an abstract design level, of the beginning of the exception-handling process.

## 3.3 General Policies regarding Exception Handling

When one or more programmers implement a system they will often handle exceptions as and when the situations occur, hence the differing standards of exception handling within a single application. General policies should rather be integrated into the development process and address the need for an integrated exception-handling policy spanning the whole software

development life cycle to counteract the problems experienced with the *ad hoc* methods used in practice.

The current approaches to exception handling follow one of three main policies: Ignore, Acknowledge-Notify and React-Repair.

### 3.3.1 Ignore (also referred to as the Novocaine) Policy

This system ignores all exceptions and no guarantees are given [46, 58, 63, 70]. The resulting system will be unpredictable. The only certainty is that it will ignore any exceptions and, depending on the severity of the exception, produce unpredictable results. It may be possible to a certain extent to predict the behaviour of the system according to empirical results. It will, however, be application- and even platform-dependent and the system may or may not be able to continue with execution. This is the easiest policy to implement, since no provision is made for exception handlers and therefore no additional coding is required. This policy guarantees reliability of level zero as mentioned in Section 3.2.

The software developed under this policy does not satisfy the requirements of reliable software.

- Example: Display of web pages. The browser ignores all code it does not understand – the worst that can happen is that the page does not load or that the browser crashes.

### 3.3.2 Acknowledge-Notify Policy

This system only acknowledges any exceptions it expects to encounter. Guarantees are given with regard to the type of exceptions acknowledged. For obvious reasons an unexpected exception cannot be acknowledged other than as, for example, with a message stating "unknown exception" and as yet it is impossible to pre-determine an exhaustive list of all possible exceptions for a system until the implementation is complete, since certain decisions and choices made during implementation and their outcome are not known beforehand [70]. These decisions and choices influence the set of possible exceptions that may arise.

This scheme is more complex to implement, since additional code is needed for the detection and acknowledgement of the exception. As with the previous approach the system may or may not be able to continue execution.

Acknowledgement can be in the form of a textual message or numerical error code (these are usually a list of numbers used to indicate and identify exceptions) displayed externally to notify the user, or by writing internally [58] to an exception log, with no external indication of an exception, or both [13]. Important issues include:

- The detection of the occurrence of an exception.
- Identification of, and differentiation between exceptions by means of a set of characteristics. Exception identification and differentiation characteristics can also be used as a basis for classification and categorisation. Due to the diversity of exceptions, provision cannot be made for each individual exception and exceptions must therefore be *classified* so that a single handler can handle any exception as part of a group to prevent redundant code.
- Acknowledgement of the occurrence of an exception, externally and/or internally, and the type of information provided and the accompanying change in flow of control – from detection to acknowledgement and if possible back to normal execution.

Supplementary code is necessary for the detection, identification and acknowledgement of exceptions. Responsibilities also need to be assigned to determine which routines will be responsible for the detection and acknowledgement, and also which routine will be responsible for monitoring the flow of control in the system. The software developed under this policy comes closer to satisfying the requirements of reliable software but does not meet all the requirements as described in Section 3.2. This policy might reach level two as depicted in Diagram 3.2.

- Example: The Pascal compiler. It acknowledges syntax errors and gives an indication to the programmer as to the existence of the error. In certain cases it makes a suggestion as to the cause of the error but does not attempt correction. (No attempt is made to detect semantic errors. Semantic errors are ignored even if they lead to infinite loops.)

### 3.3.3 React-Repair Policy

The system that uses this policy not only detects and acknowledges the exception but also reacts in order to effect a repair. Different levels of guarantees can be given, depending on the system – for example, it could commit-and-rollback, or exit gracefully. This is the most difficult policy to implement, because not only is additional code needed for the detection and acknowledgement of exceptions but code is also required for the reaction of the system to the exception as it attempts a repair. In addition to the issues surrounding the *Acknowledge-Notify* policy there are the actual handling strategies, which include strategies for dealing with

unexpected exceptions. The course of action upon failure of a handling mechanism also needs to be determined. This is also the most expensive policy due to more code, and therefore more development time being necessary than is the case with the previous two policies. Responsibilities must be assigned to the routine for dealing with the detection, acknowledgement, handling and repair, overseeing the process and controlling the flow of control. The software developed under this policy can satisfy the requirements of reliable software as mentioned in Section 3.2 and in general will deliver level four guarantees as depicted in Diagram 3.2.

- Example: Windows Software. When the software cannot find a file pointed to by a shortcut, it then searches for that file through all the local discs in order to try and locate the specific file.

### 3.3.4 Hybrid Policies

Hybrid policies are also possible where a combination of the above can lead to more reliable software under certain conditions - especially where the application of only one policy might not be feasible throughout the system's development. However, a hybrid policy should be implemented with care to avoid the pitfalls of having no policy at all since it is less uniform than those mentioned in Sections 3.3.1 – 3.3.3.

All the above-mentioned policies need to be formulated and applied before they can be of any use. To implement them, strategies are necessary – these strategies must deal practically with the very real issues mentioned in the previous Sections 3.3.2 and 3.3.3.

## 3.4 Strategies for Policies

As Millewski [50] says *"The error-handling strategy is one of the most difficult and visible problems in professional software"*. The author argues that the single most important requirement is that an exception-handling policy must be applied uniformly throughout the system [4, 9, 46, 58, 71]. Well thought through strategies are needed to make the two major policies of Sections 3.3.2 and 3.3.3 operational, and to address the related issues.

The existing design-level strategies to deal with the different issues within exception handling will be now be explored. In order to apply the policies of Acknowledge-Notify and React-Repair, exceptions must first be detected. Exception detection will be discussed in Section 3.4.1. Exceptions that occur in systems are not all the same, and it is impossible to write individual handlers for each and every exception, thus exceptions must be identified and

grouped accordingly in order to assign handlers to these groups as described in Section 3.4.2. Handlers dealing with these exceptions are located at various locations and are called when an exception occurs. This means that the flow of control is changed as mentioned in Section 3.4.3. Exception propagation is related to the change in flow of control, i.e. the passing on of the handling of an exception if the current handler is unable to handle the exception, and responsibility assignment, i.e. who is responsible for what and when. These two matters are dealt with in Sections 3.4.3.1 and 3.4.3.2. The different handling models are discussed in Section 3.5. Current language-specific implementation details will be discussed in Chapter 4.

### 3.4.1 Detection of Exceptions

*"The first step in detecting exceptions is, of course, to have some model of the "correct" behaviour both for the entire system, as well as for each individual component"* [18]. Before any exception can be dealt with, it must first be detected and measured against a model of the proper functionality of the system [18, 37]. Most exceptions cannot be explicitly detected but are implicitly detected due to the effects they have on the system [53].

For example, if a character value is entered instead of an integer it can cause an exception when an arithmetic operator is applied. User input should be tested for the correct type before the operator is applied so that the erroneous input is detected as being a violation of a constraint [9] – in this case that it should have been an integer and not a character. The exception was not detected when it was entered, but only when it was tested for violation of a constraint.

In order to detect an exception it must be related to a concrete construct in the software system. The behaviour of the system is compared with the specified "correct" behaviour for deviations [12, 46]. In the previous example two problem areas can be identified - incorrect input values, which pass the constraint tests, and incorrect or missing constraints allowing incorrect values to pass:

- If a value is entered within an allowable range, but is incorrect, the system will not be able to detect it because it cannot be compared to a concrete construct within the routine. For example, if a quantity of items ordered is entered as "5" instead of "6" there is no constraint violation.
- The other side of this scenario is the situation where the constraint is incorrect or missing [41], in other words where the test for validity is incorrect or absent. Issues regarding missing and incorrect constraints were dealt with in Chapter 2.

In most cases within the system, exception detection is related to violation of constraints derived from the pre- and post conditions of routines.

Two detection modes can be implemented, namely active or passive. An active detection mode will perform periodic checks for violation constraints while the passive detection mode waits until an exception occurs [4]. The active mode is inefficient, as it requires resources to perform the periodic checks while this overhead is not required by the passive mode. However, more resources will be necessary to resolve exceptions detected by the passive mode than the active mode as the latter mode lends itself to the implementation of preventive measures. Thus, at the time of resolution the overhead requirements will be reversed.

The following sections describe different methods of relating the occurrence of an exception to concrete constructs in the program or routine, in order to detect the possibility of the occurrence of an exception.

#### 3.4.1.1 Violation of Constraints [8, 9]

One method of detecting exceptions is to associate all exceptions with violations of constraints [9]. Constraints are restrictions, limits or boundaries through which validity can be established by testing values for violations of these constraints. The constraints in most systems refer to the pre- and post conditions as well as invariants. Pre-and post conditions refer to constraints that must be adhered to before and after the execution of a routine, while invariants refers to conditions that must hold indefinitely to ensure successful execution. Error prone values can then be explicitly tested in order to establish if they adhere to the requirements of the pre- and post conditions and invariants and if not, the values will then be known to generate an exception.

#### 3.4.1.2 Contract Theory [47, 52]

This method is an extension of the above method, whereby the notion of a contract is added and is used mostly in concurrent and distributed systems where routines communicate via messages. The messages are exchanged by routines requesting services (clients or callers) from other routines (servers or callees). A contract is entered into between a client and a server and specifies how to use a routine and what can be expected from a routine [52]. The extension makes provision for the uniform implementation, in centralised as well as distributed systems. Mandrioli and Meyer [47] see exceptions as contract breaches between clients (requesting services), from the servers (service providers). The called routine either fulfils, or fails to fulfil, the contract, with no other alternatives. In this scheme the client is

responsible for ensuring that the preconditions hold at the time of requesting the service from the server and the server must ensure post-conditions on replying to the client.

Ensuring pre (or post) conditions can be treated in two ways:

1. either it must be provable that the precondition will always be satisfied, or
2. that it can be embedded in a conditional construct.

(This is not always possible, and even if it is possible, it is not always practical to express all preconditions in this way [47]). A violation by either the caller or the callee must be easy for the other to detect in order for this scheme to work.

Three problem areas arise [47]:

1. Operations that can only be evaluated by attempting execution.
   (For example, to determine whether the sum of two numbers will cause arithmetic overflow on a certain machine.)
2. Frequent operations with a small probability of failure.
   (For example, basic arithmetic operations for which the result must fit in the specific machine's number system.)
3. Software fault tolerance.
   (For example, a programming/logic error through oversight by a programmer.)

The three problem areas cannot be dealt with uniformly, and thus special provision must be made for each of them. The first case can be addressed by embedding such an operation in a conditional construct. An attempt can be made to solve the second problem by proving correctness. The third is application and platform dependent.

Once exceptions are detected, they can either be acknowledged and/or handled, depending on the policy chosen from Section 3.3. Making exceptions detectable is only the beginning of the process of successful exception handling. After detection, exceptions must be classified in order to be handled – as mentioned in Section 3.3.2. This will be discussed in the following section.

### 3.4.2 Identification and Classification of Exceptions

There are many different types of exceptions. In order to manage exceptions, related exceptions can be grouped [58], depending on their characteristics, in order to simplify and unify the process of exception handling. For example, the following four exceptions could occur:

1. Division by zero.
2. Calculating the power of the base number zero with an exponent of zero.

3. Writing to a file that does not exist.
4. Insufficient memory.

The first two exceptions are related and can be grouped together as computational exceptions, whilst the third can be classified as an exception originating from a logic error and the fourth as an exception due to a hardware error.

The identification is related to the naming of exceptions. This is dependent on the application and can be derived from the type of violation constraint or service requested that couldn't be provided. Error or return codes [13] are, for example, one method, whereby each exception is related to a unique number that identifies it. Naming exceptions should be done uniformly and this becomes especially important when an exception is reported. Chapter 5 contains more information on this topic.

As the above examples illustrate, exceptions that occur in programs are diverse. Distinctions should be drawn between the different exception types in order to categorise them. Categories can then be handled in a uniform way in order to streamline the process. It could possibly be more efficient to handle all related exceptions with the same handler since it would be wasteful to write a different handler for each exception (as will be discussed in Section 3.4.3).

Many different exception classifications can be found in the literature, often formed from different perspectives. These perspectives range from general to specific, sometimes 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. (Possible causes and elimination possibilities have been discussed in Chapter 2.)

#### 3.4.2.1 General Classifications

The first three classifications discussed in this section are very general. According to Dickson *et al.* there are two broad types of exceptions [15], expected and unexpected. Expected exceptions are those that are explicitly planned and provided for. Unexpected exceptions are simply described as unanticipated. Duesing and Diamant make another general classification in their work with the C++ SoftBench CodeAdvisor [22]. They describe exceptions as if belonging to two sets:

1. The one set constitutes the set of all real exceptions.
2. 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 3.3.

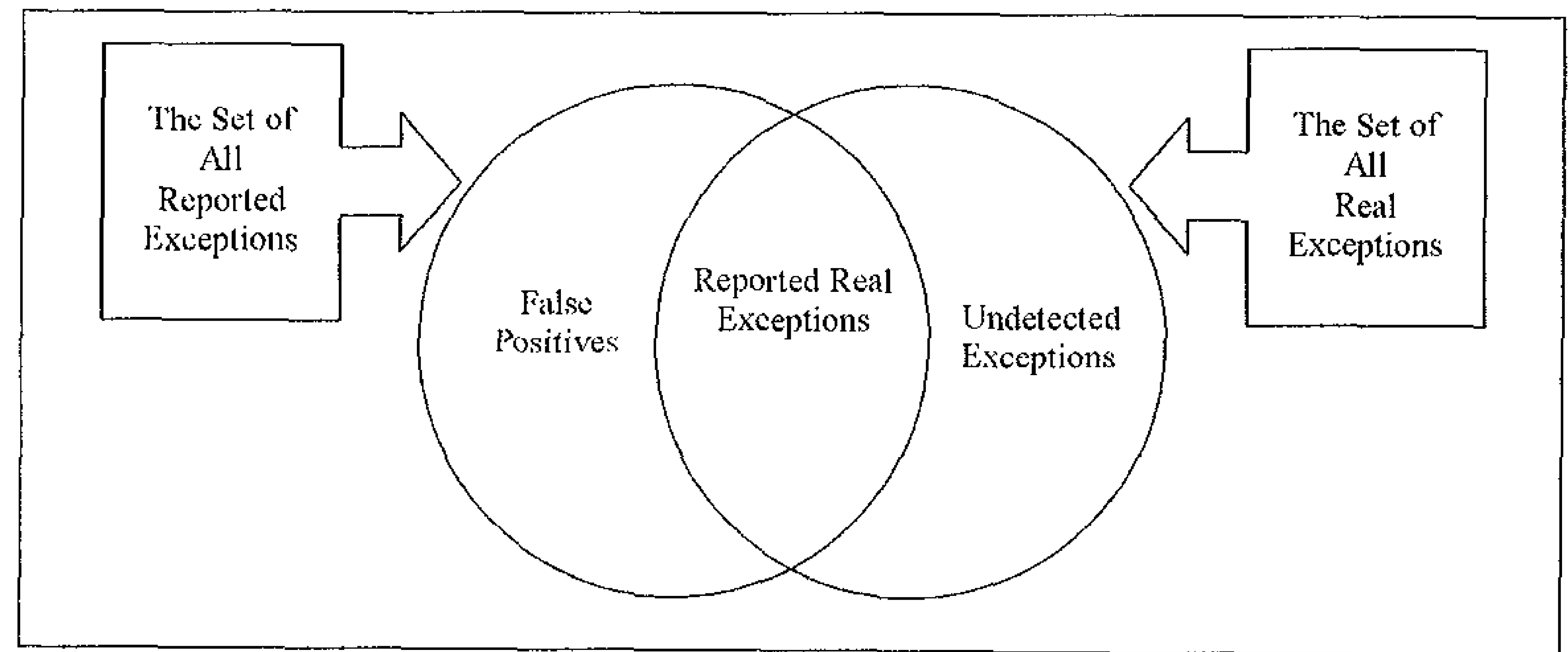



**Figure 3.3 – The classification of exceptions according to Duessing and Diamant [22].**

Another general analysis of exceptions by Luo *et al.* depicts exceptions on three orthogonal axes [46]. The labels given to the different axes are:

- Detectable.
- Resolvable.
- Known.

Detectable exceptions are those that a system is capable of detecting, 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 solution *when the exception occurs*. Known exceptions refer to those the system currently has knowledge of, unknown meaning those beyond the system's current knowledge. Current knowledge is limited and extended through detection of previously unknown exceptions. The lengths of these axes extend as the number of detectable, resolvable or known exceptions increases, as depicted in the Figure 3.4.

All detectable and resolvable exceptions are known. Detectable and resolvable implies that some form of provision is made for the exception, and this cannot be done unless it is expected and thus known. The opposite is not true. Known exceptions might not all be detectable as detection measure may elude developers. Even if detectable, they might also not be fully resolvable.

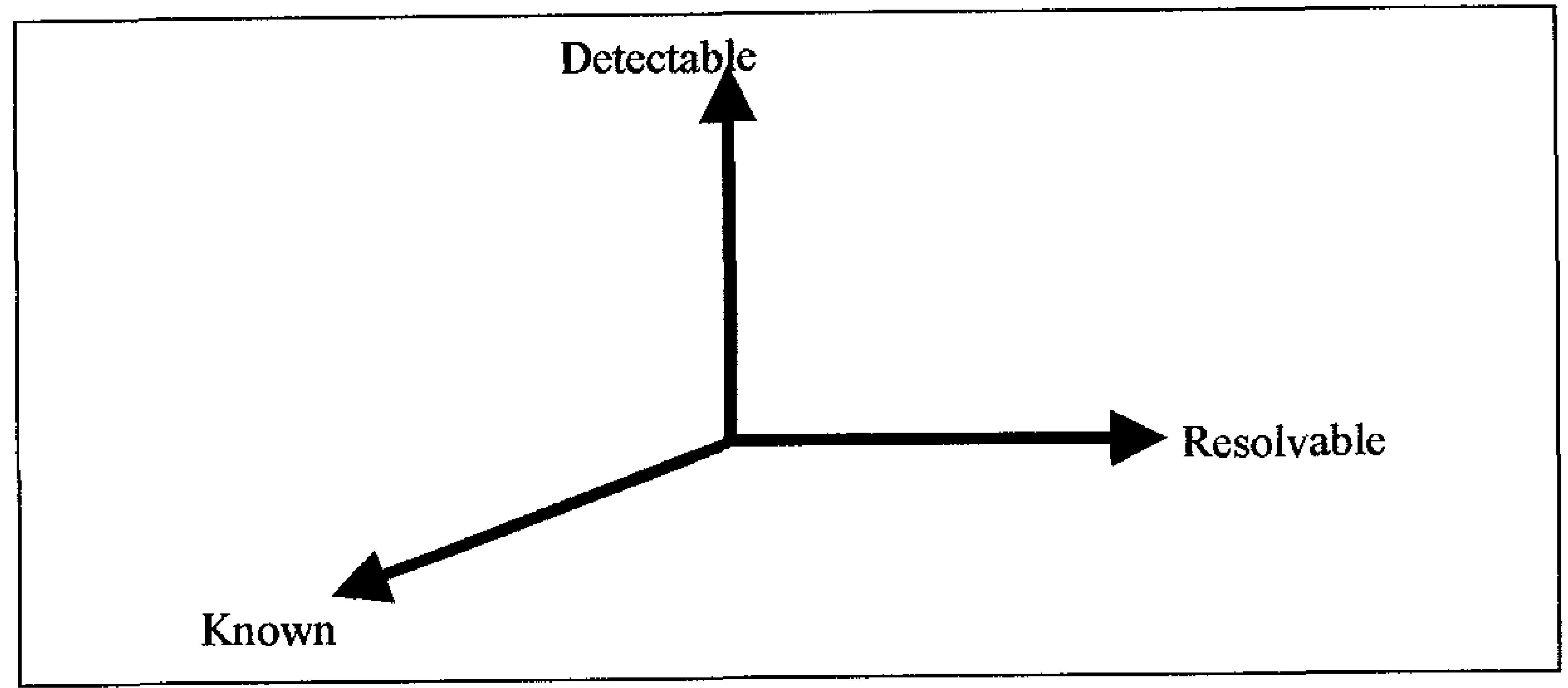



**Figure 3.4 – The classification of exceptions according to Luo *et al.* [46].**

These classifications are very general and may aid the compilation of general, high-level abstract policies. They can be related in the following way as depicted in Figure 3.5:

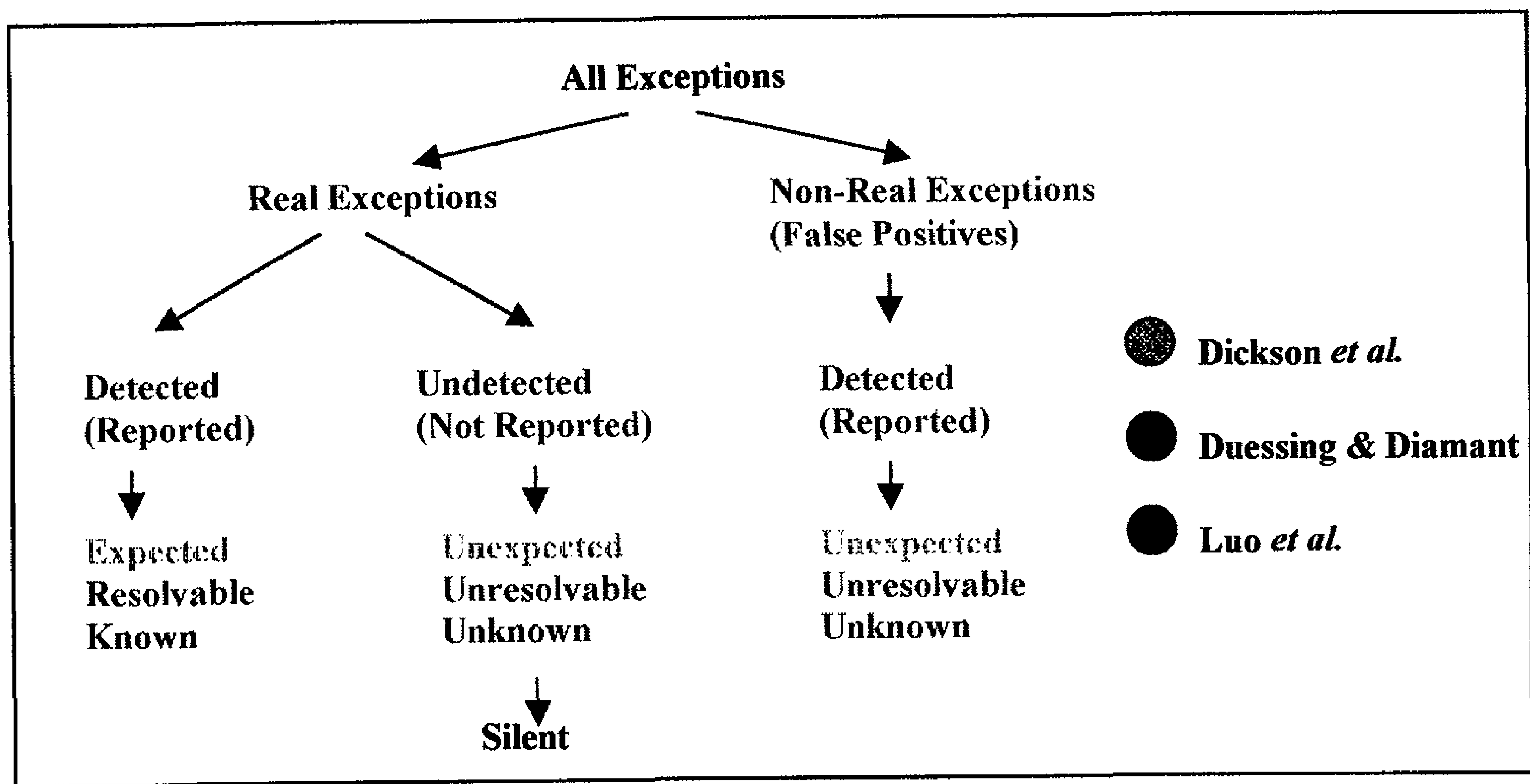



**Figure 3.5 – The relationship between the general classifications.**

### 3.4.2.2 Classifications According to Causes and Origins.

The next four classifications are more specific. They relate to lower level constructs and use the causes and origins of exceptions as a basis for classification.

Mandrioli and Meyer formed the following classification of exceptions with reference to their experience with Eiffel systems in practice [47]. 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.

Garcia *et al.* classifies exceptions into three classes according to the origin of the exception in the code [26].

1. Interface exceptions – signaled when conflict arises in the interface of a software routine when the input provided is different from what is expected.
2. Failure exceptions – signaled when the services requested cannot be provided, in other words, the routine is unable to complete execution due to an exception external to it.
3. Internal exceptions – raised by the routine whenever an exception occurs internal to the routine.

Bail also makes use of the possible origins of exceptions as a basis for classification [4]. He states that there are two possible causes for exceptions –

- Software defects.
- Erroneous data.

This corresponds to the classification of program errors and data errors as given by Dony [20]. Software defects refer to the errors in the code's logic. This may be caused by 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.

These classifications can be related as depicted in Table 3.6. The arrows indicate the correspondence between the adjacent classifications. There is therefore a mapping from any one of the classifications to any other. The fact that they can be related is important, as it indicates that there is more correspondence at a lower level than the previous section's classification and it may be an indication that these classifications form the basis of a universal classification scheme.

| Causes | Origin | Origin | Origin |
| --- | --- | --- | --- |
| Mandrioli and Meyer [47] | Garcia *et al.* [26] | Bail [4] | Dony [20] |
| Assertion Violation | Interface Exceptions | Erroneous Data | Data Errors |
| Routine Call Fails | Failure Exceptions | | |
| Routine applied to Non-Existing Object | Internal Exceptions | Software Defects | Program Errors |
| Hardware or O/S Exception | | | |

**Table 3.6 – The relationship between the four classifications.**

### 3.4.2.3 Classification According to Severity.

A defect classification framework by CSST Technologies classifies exceptions according to the severity of the results of the exception [16]:

- Critical – the failure of the complete software unit, subsystem or entire software system.
- Major – the result is the same as for the above, but in this case there may be alternative options that can be implemented to circumvent the failure.
- Average – the result is not failure but the system's results could be questionable or the system's functionality might be degraded.
- Minor – the correct processing might be obtained with some minor exception handling.
- Exception – due to non-compliance and it might be ignored or postponed.

This classification is very different from the previous ones and should not be used in isolation. It may be useful to establish the possible severity of the effect of an exception in addition to the classification in Section. 3.4.2.2 to ensure that it is dealt with properly, especially when the effects on the environment are critical.

### 3.4.2.4 Comparison

Table 3.7 is a summary of all the exception classifications, with the colours between the different classifications indicating correspondences. For some classifications more than one classification corresponds to the same heading in another classification; the mapping is thus not one-to-one.

| General Classification | | | Causes | Origin | |
|---|---|---|---|---|---|
| Dickson *et al.* [15] | Duesing and Diamant [22] | Luo *et al.* [46] | Mandrioli and Meyer [47] | Garcia *et al.* [26] | Bail [4] and Dony [20] |
| | Reported | Known | Assertion Violation<br>Routine Call Fails | Interface Exceptions | Erroneous Data |
| Expected<br>Unexpected | Real<br>Undetected<br>False Positives | Resolvable | Routine applied to Non - Existing Object | Failure Exceptions<br>Internal Exceptions | Software Defects |
| | | Detectable | | | |

**Table 3.7 – Comparison of classifications.**

An exception will be classified during the earlier phases making use of the more generic classifications, i.e. the first three and during later phases by making use of the more specific classifications, i.e. the last three. This accords with the software development process whereby the system is specified firstly in terms of abstract or general ideas, followed by more specifics later. The classification of Section 3.4.2.3 is not included since the criteria crosscut all the different categories, depending on the circumstances.

### 3.4.2.5 Integration

The classifications mentioned earlier need to be integrated into a single classification to be of use in managing the vast exception possibilities that can occur. As mentioned before, exceptions cannot be handled individually and should be grouped in order to be handled efficiently. The classification will ultimately not aid in the resolution of the exceptions but in the appropriate assignment of handlers.

Each classification has definite merits and is different from the others but it would be helpful if a correlation between them could be found in order to establish an integrated classification. To satisfy this need a high-level integrated classification of exceptions has been derived and is shown in Figure 3.8. The graph depicts the relationship between each different type of exception and three criteria – the predictability, detection and resolvability of exceptions. Predictability refers to the ability to establish beforehand whether an exception can occur in a system. Detectability refers to the ability of the system to notice the occurrence of an exception and resolvability to the ability to provide a solution in one form or another. These are the most important criteria for the classification of exceptions since the life cycle of exceptions revolves around these criteria. The life cycle of an exception revolves around the detection, propagation and handling of exceptions [13]. The detecting and handling of an exception depends on the predictability thereof, and in order for it to be detected and handled measures must be put in place to detect and handle the exception on the grounds of it having been predicted. Propagation relies on the provision of handlers, which try to correct behaviour and thus resolvability

The origin of the axis system depicts the value (resolvable, predictable, and detectable) whereas the extreme ends of the x, y and z-axes depict the unresolvability, undetectability and unpredictability of exceptions, respectively. The further along the respective axis it lies the more undetectable, unpredictable and unresolvable it becomes.

"Silent" and "Reported" exceptions are opposites: "Reported Friendly" and "Reported Unfriendly" are distinguished since the user must always be considered in reporting exceptions. The former implies that a useful exception message will be generated so that the user will have a clear idea as to what the cause of the exception was and what measures should be taken in order to resolve the problem. In the case of the latter the message will contain less helpful information due to the circumstances surrounding the detection of the exception or because it is simply not reported correctly. Silent exceptions are real exceptions, which pass through the system unnoticed. They pass undetected and are thus unresolvable and unpredictable.

A user-friendly message will only be generated if the exception is detectable but this is not dependent on the resolvability of the exception. To prepare a handler for the generation of the message implies that the exception must be detectable by means of certain conditions. The message can be adapted to reflect the fact of the exception being resolvable or not. In the case of an unpredictable exception a handler could generate a standard message. This means that

although it was not expected, general provision has been made for unexpected, detected exceptions. The generated message will be a general one and thus less informative.

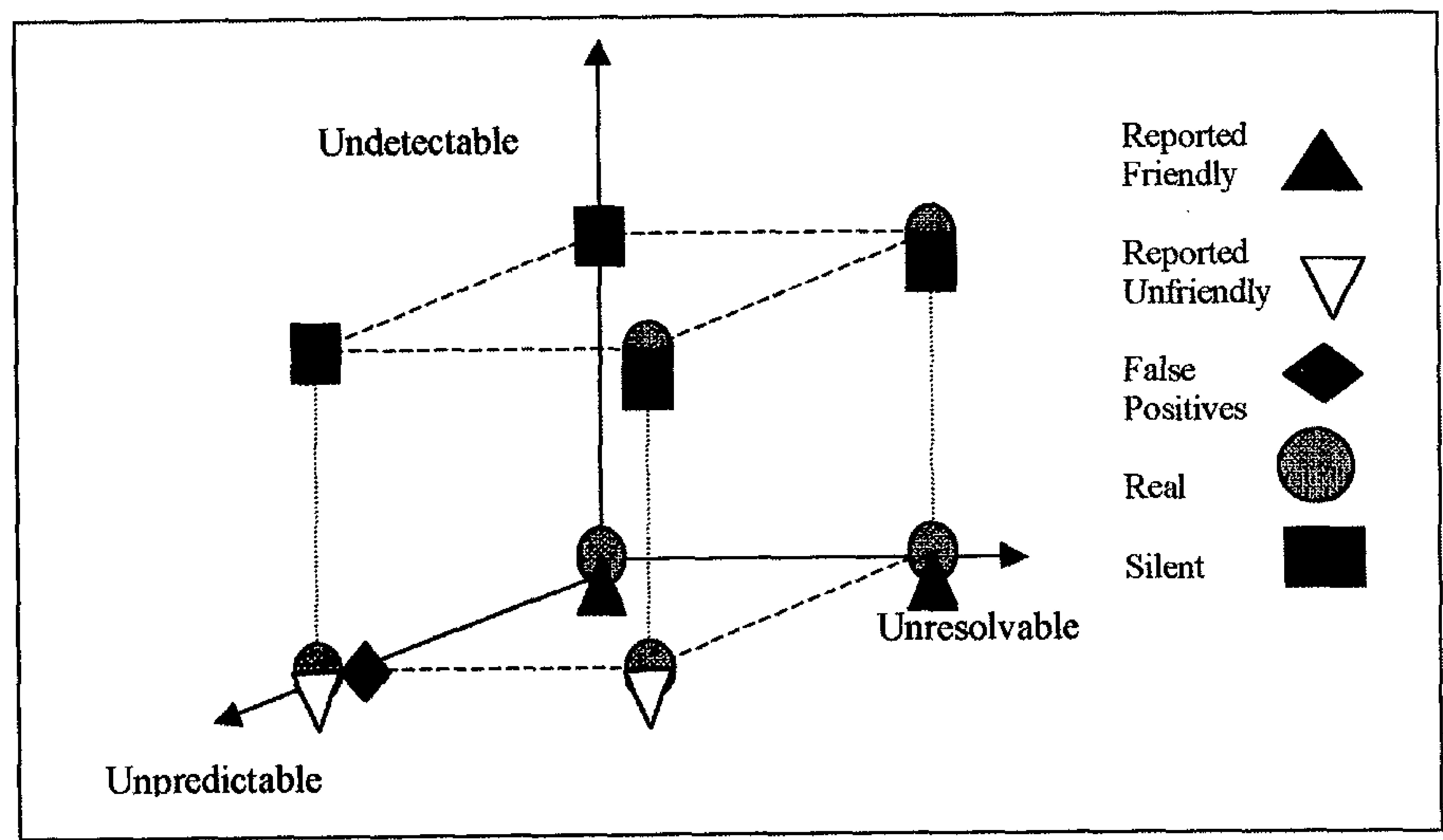



**Figure 3.8 – Integration.**

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 nature. Their existence is only established through their 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. 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 generated but in the case where they go undetected no message can be generated and the situation will also not be resolvable since it is undetectable.

Figure 3.8 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 3.8 as related to all the previous classifications each of the

classifications can be positioned on the diagram. The legend of Figure 3.8 is used to code the position of each of the classifications in Table 3.9.

| General Classification | | Causes | | Origin | |
|---|---|---|---|---|---|
| **Dickson *et al.* [28]** | **Duesing and Diamant [14]** | **Luo *et al.* [46]** | **Mandrioli and Meyer [3]** | **Garcia *et al.* [35]** | **Bail [27] and Dony [12]** |
| Expected ▲● | Reported Real ▲● | Known ▲● | Assertion Violation ▲● | Interface Exceptions ▲● | Erroneous Data ▲● |
| | | | Routine Call Fails ▲● | | |
| Unexpected ▽● | Undetected ■ | Resolvable ▲● | | Failure Exceptions ▽● | |
| | | | Hardware or O/S Exception ▽● | | Software Defects ■ |
| | False Positives ◆ | Detectable ▲● | Routine applied to Non-Existing Object ▽● | Internal Exceptions ▽● | |

**Table 3.9 - Application of the classification.**

For each of the items of the classifications the type of exceptions can be expressed in terms of being real or not, being reported or not, and if reported, reported in a friendly way or not.

Once exceptions are detected and classified, acknowledgement and handling follows as described in Sections 3.4.3 and 3.5. There must be an agreement between the calling routine and the called routine on the post-exception processing [58]. The flow of control in the routine where the exception was found will change to provide for the appropriate action. The flow of control will be passed to where the exception-handling code is contained – either in the callee or caller.

### 3.4.3 Flow of Control

Change in the flow of control alters the flow of control from the normal processing flow to that of exception acknowledgement and handling, irrespective of the physical location of the code. Since there are too many types of exceptions to be able to handle each individually some of the exception handlers have to be either reused or redundantly coded. Reuse implies that the handler is used by different routines to handle the same type of exception. In this case it implies that the handlers will not be local to all exception occurrences and that for some exceptions the control will have to be passed to a handler embedded in code elsewhere. Redundancy implies that an identical handler is repeatedly coded for each routine that might need it.

There is an ongoing debate about whether to distribute exception-handling code throughout the system, favouring redundancy, or whether to keep the code separate, enabling reuse as a separate exception-handling object [7, 79]. The advantages of centralising the code outweigh the disadvantages. Table 3.10 lists the advantages and disadvantages of each approach:

| | **Advantages** | **Disadvantages** |
|---|---|---|
| **Centralising the exception-handling code** | [4, 7, 26, 37, 47, 51, 57, 58, 79]:<br>• Increase maintainability.<br>• Apply uniformly.<br>• Read and comprehend because the algorithm is free from exception-handling code.<br>• Eliminate redundancy of code by allowing the programmer to implement the reaction to different occurrences of the same exception in a single place.<br>• Reuse of code possible. | [72]:<br>• Adds computational overhead from regulating the flow of control to and from different locations.<br>• It is difficult to report context from code external to where the exception occurs. |
| **Distribute the exception-handling code** | No performance is lost due to the regulation of the flow of control in a system where the exception-handling code is distributed to the locations where it is required. | The code becomes more difficult to:<br>• Maintain.<br>• Read and comprehend.<br>• Reduce redundancy of code. |

**Table 3.10 – The advantages and disadvantages of centralising and distributing the exception-handling code.**

To change the flow of control to the handler, thus referring the handling of the exception to another part of the code implies that responsibilities must be assigned to different parts of the code for the detection, notification and handling of the exception.

The deference exception handling by the immediate section of code to another part of the code is called exception propagation and will be discussed in Section 3.4.3.1 and the issues surrounding responsibilities in Section 3.4.3.2.

#### 3.4.3.1 Exception Propagation

This is especially useful in nested executions. The term exception-neutrality indicates that *all* exceptions are referred to the caller [77]. It allows for the nested routine to complete and raise the exception in the containing routine [71, 84, 86]. It can also be implemented by placing a partial order on handlers as if a tree-type structure whereby an exception handler higher in the hierarchy can handle all those below. If no appropriate handler can be found the exception is propagated to the universal exception handler that catches all exceptions uncaught and unhandled [4]. An exception list determines those exceptions that will be allowed to propagate to their callers. It is created as part of a routine's signature and exception propagation will only be allowed if specified by the exception list [13]. After successful handling, control can be returned to the routine calling the handler for continuation of execution.

Two modes for propagation are mentioned - automatic and explicit [26, 86]. Automatic propagation implies that if no immediate handler is found, then the exception is propagated to a higher level in the hierarchical structure, in other words the caller's callers, which can result in loss of context. Explicit propagation implies that the propagation is limited to the immediate caller. The handlers can be allocated statically or dynamically [13, 38, 85]. Static allocation means that the exception and handler binding is static and predetermined while in the case of dynamic allocation the handler for the exception is determined from criteria at the time of the occurrence of the exception. The following criteria can be used for dynamic allocation to choose between different applicable handlers [13]:

- Location - The handler located closest to the exception.
- Specificity - The more specific handler for the exception.
- Agreement – The best match handler for the exception.

These criteria can also be prioritised to determine which criteria will be the most important [13]. Static allocation decreases the reuseability of the code but on the other hand it is easier to implement than dynamic handling, which also requires more processing and thus more overhead [38].

Should an exception handler have an exception handler in case it fails? This can lead to the provision of an infinite number of exception handlers. Should recursion be allowed for failing handlers? Mandrioli and Meyer claim that the exception handler should never fail or cause an exception – it should be simple and easily verifiable and assumed to be safe [47]. If this is accepted it is decided that in the very unlikely event that an exception handler fails no guarantees can be given.

#### 3.4.3.2 Responsibility Assignment

Responsibility needs to be assigned to routines as to the detection, propagation and handling of the exception condition [4]. Correct and consistent responsibility assignment can avoid missing and duplicate exception-handling mechanisms. Inconsistent responsibility assignment can lead to failure of exception-handling mechanisms due to inconsistent assumptions between routines (clients and servers), different views on default decisions and incompatible conventions [4]. The earlier in the design responsibilities are assigned the more cohesive and consistent the exception-handling behaviour will be, it will ensure that the different components share the same view on the handling of exceptions [4]. A policy can ensure that consistent views are held on default assumptions, and also ensure the compatibility of conventions [4]. It is much easier to implement and maintain responsibility assignment according to defaults. Responsibilities need to be assigned at two levels, the architectural design level and the detailed design level. The former specifies the handling across components and the latter within components [4].

The following responsibilities must be assigned [4]:

- Detection – "which routine notices the exception"

  There must be clarity on which routine's responsibility it is to detect an exception. The responsibility for the detection of the exception normally lies with the current routine. When it is called, a precondition check should be performed in order to establish that any preconditions are satisfied. If no constraints are violated and the post-conditions are satisfied the routine has executed successfully and met all obligations. Otherwise an exception occurred.

- Propagation and notification – "which routine informs other routines and when"
  The routine that is responsible for the detection must also be clear on the notification procedure that follows - which routine is responsible for notifying which other routine(s). The routine in which the exception is detected must notify the caller and pass the responsibility for handling the exception to the handler, thus signalling and informing the caller of the exceptional condition [4]. At this point the flow of control is altered to make provision for the exception that has occurred.
- Handling and recovery– "which routine performs handling, how and where"
  No ambiguities must exist in assigning responsibility for the handling of an exception to a routine. The exception handler for the specific exception is invoked to handle the exception [4]. At this point the handler takes over the flow of control from the calling routine. After the exception is handled the flow of control is changed again in correspondence to the specification of the handling model implemented. Again a set of rules must exist so that each routine knows whom to notify.

Once the responsibility is assigned the responsible party must take appropriate action when called upon. The handler can take any of the courses of action described in Section 3.5.

## 3.5 Handling Models/Techniques

Exception handling is described as follows:

*"Exception-handling is a programming language control structure that allows the normal execution of a program to be replaced or augmented by special exception-handling code when certain special events or conditions occur"* [9].

Exception handling is thus the ability to logically separate the *reaction to an exception* from the *location* where the exception occurred [51]. The handling of exceptions leads to the development of dependable software systems by incorporating the process of recovery from exceptions into the system [17].

In general exception handling is constructed around the following three basic concepts [51]:

- A block of related statements that is protected against exceptions, referred to as a "guarded block".

- One or more exception handler(s) specified for the protected block.
- A mechanism to raise exceptions.

During the occurrence of an exception in the protected block an exception is raised and the corresponding handler executed. In general, a "block of code" implies multiple statements combined to form a single functional unit [13].

Exceptions can be handled depending on their severity and the implications for the system. A trivial exception can be ignored or excused while an input error can be retried leading it to be more permissive than restrictive [15, 54].

Furthermore a decision needs to be made with respect to which of the following modes could be applied [15]:

- *Trivial exception handling* applies usually to external and expected exceptions. If such an exception occurs complex exception handling is not necessary.

  For example, erroneous input from users, whereby the system responds with a short message and the user is prompted to re-enter.
- *Automatic handling*, whereby the system takes automatic action in order to handle the exception. When automatic handling is used it must be predetermined whether the operation is optional, critical, repeatable or replaceable.

  For example, if a file is written to that does not exist the system can create the file.
- Through the process of *co-operation* between the system and the user to handle the exception [26].

  For example, prompting the user for options to recover.
- *Manual recovery*, whereby a human takes responsibility for the handling of the exception [26].

  For example, when a file is read-only and must be written to, the user can change the properties associated with the file.
- *Failures*.

  If none of the above can be applied and the exception cannot thus be handled there needs to be a policy about how this situation will be dealt with too.

The decision on which model to choose depends on the assessment of the extent of the damage resulting from the exception [53]. These models are based on what code they execute *after* the occurrence of an exception. The different handling models are described below:

### 3.5.1 Non-Local Transfer Model [13, 86]

A routine specifies a point of transfer with, for example, a "goto" statement, a location elsewhere where the possible handling can take place. The routine also contains a pointer to an activation record on the stack containing the transfer point. It is very general and allows branching anywhere in the program which makes it difficult to maintain. This model does not clearly define how or at what point the execution continues after the exception is raised and handled.

### 3.5.2 Termination Model [8, 13, 20, 26, 40, 47, 49, 51, 69, 85]

The termination model terminates the block of code containing the exception and executes the first statement after the block. This model assumes that the called handler completes the execution of the system to a termination point [47, 84]. It is as if the incomplete operation within the block completed without an exception – the handling mechanism replaces the faulty operation with an alternate [13]. If the current handler cannot complete the execution it is propagated until a handler is found that can complete the execution to point of termination. Termination implies that just that - termination -and resumption is not allowed [27]. Diagram 3.11 illustrates the mechanics of this model:

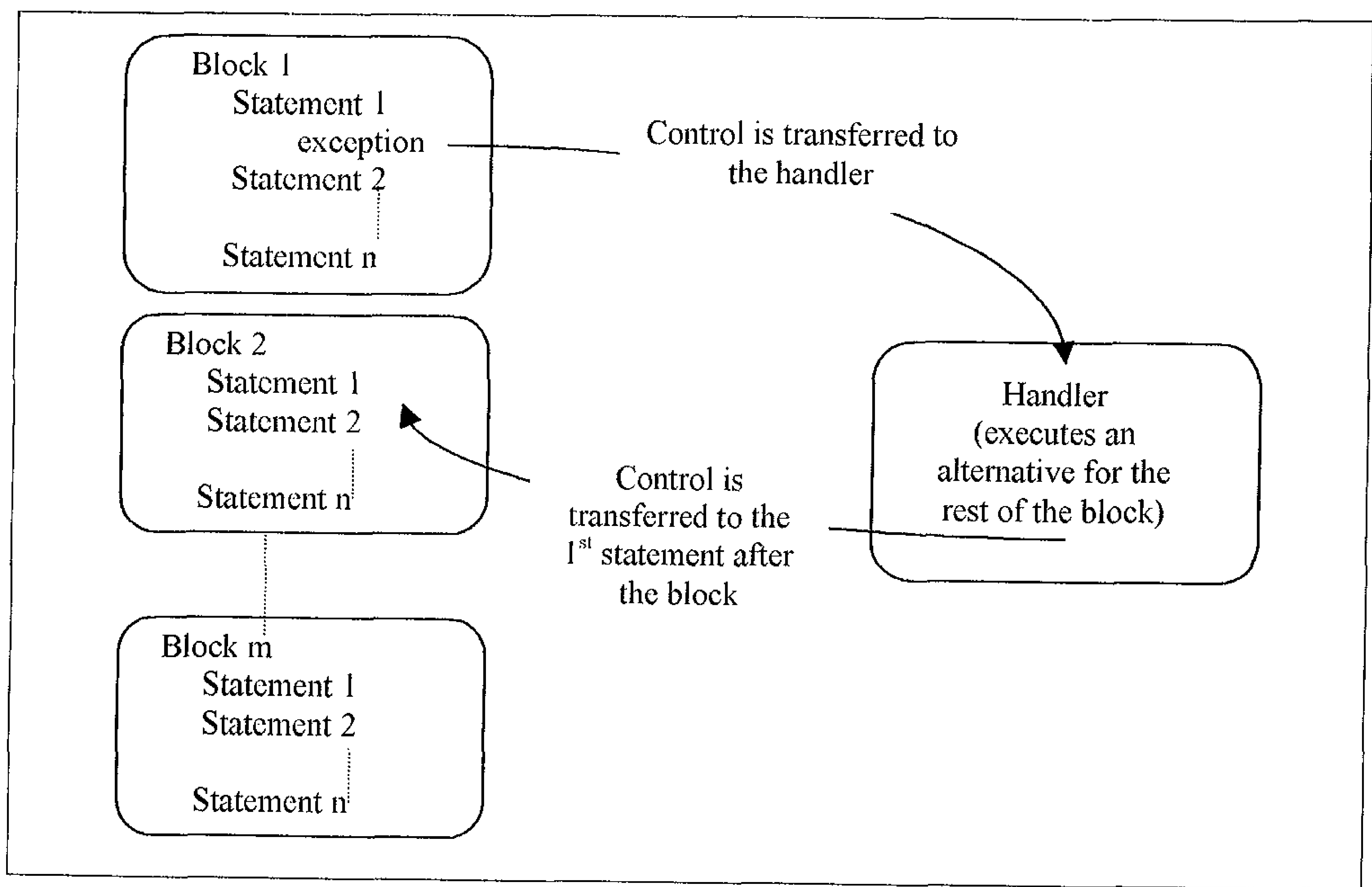



**Diagram 3.11 – Diagrammatical representation of the termination model.**

### 3.5.3 Resumption Model [8, 13, 26, 40, 47, 49, 51, 69, 85, 86]

The execution is resumed in the block of code after the point where the exception was raised. Control of flow transfers from the point where the exception was raised to a handler to correct the faulty operation and then to return control to the point where the exception was raised [13]. Resumption also implies that termination is forbidden [27]. Resumption can be made possible in the following ways:

- *Resumption – Ignore* [4, 46]

  Ignore the exception and continue with the next line of executable code. Although this is a valid option it may not always be possible [9]. A variation on this model is where the exception is ignored but recorded in an internal log.

- *Resumption – Excuse* [4, 8, 9]

  Excusing non-crucial values is also a possibility. Strict control must be exercised on the values allowed to be excused [8]. If, for instance, the age of a person is not of cardinal importance for the rest of the execution an incorrect value or value of incorrect format can be "excused" - thus be allowed without creating an exception and continuing with the execution of the rest of the code A safer alternative to saving the incorrect value or the value in incorrect format is to substitute the value with a default value in a default format, which, although not absolutely correct, will at least not be the cause of further exceptions [8]. Such a default value, although propagated, will eliminate the need for caution or further testing, as it will be chosen so as not to cause exceptions [8]. A message to this effect might be generated to inform the user that the exception has been "excused" or "substituted". Care must be taken not to distribute the excused information throughout the system if it can give rise to more exceptions [8]. Furthermore, an object can be created for recording such excused information for record purposes [8, 9] and it is advisable to do so. Although it model-matches the inconsistencies of reality the consequences of this technique must be well thought through before an attempt is made to implement it. Since exceptions are detected by violation of constraints and constraints must be exactly defined [8, 9] with this model the definition of such constraints can no longer be precise. The acceptable values, domain, and the range of the operation, the range, and the effects increase [27], and it adds processing overhead due to replacing excused values with default values and the control structures involved in this process [8].

- *Resumption - Forward Exception Recovery* [4, 26, 53]

  Forward exception recovery alters the state of the objects from an incorrect to a correct state making use of redundant data and analysing current states and data. It is thus

possible to continue the execution with the next line of code, as if the exception had not occurred. It is more complex to perform, needs comprehensive information about the exceptions and is application dependent [4, 84]. The user needs to be informed so as to synchronise the expectations of the user with those of the system. The result of forward recovery is not necessarily the same as the result that would have been obtained by normal processing [65]. Forward recovery is also made by making decisions on behalf of the user, and the decisions made might differ.

- *Resumption – Graceful Degradation*

  In the case of graceful degradation [4], the system is modified to continue execution with limited functionality. The functionality influenced by the exception may no longer be available, but it might be preferable to having no functionality at all. The reduced functionality should be brought to the users attention with an appropriate message.

Diagram 3.12 illustrates the mechanics of this model:

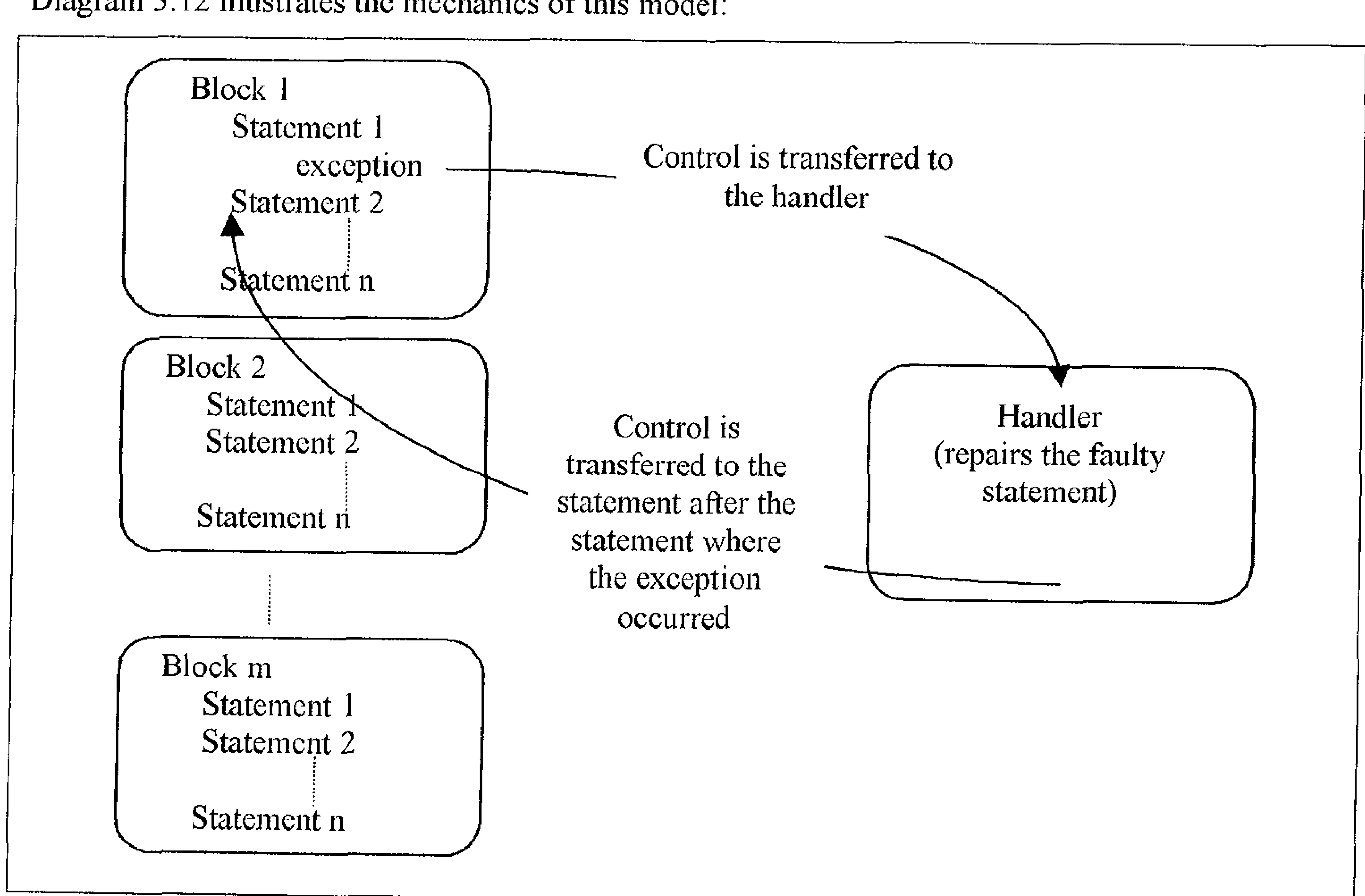



**Diagram 3.12 – Diagrammatical representation of the resumption model.**

### 3.5.4 Retry Model [13, 46, 51, 86]

The block of code in which the exception occurred is re-executed from the beginning after the exception handler repairs the exception occurrence. It restarts a faulty operation after a handler performs recovery by eliminating the conditions responsible for the faulty operation in order to attempt to repeat the operation to obtain successful execution [13]. The restarting point must be clearly defined and it is usually defined as to be the beginning of the block of code.

- *Retrying - Backward Recovery /Commit and Roll-Back* [4, 26, 46, 53, 85].

  Backward exception recovery returns the objects to the state before the interaction, [4, 29, 71] as if the exception never happened and is in general initiated by the system. The code can then be re-executed from the beginning. This is done with the aid of redundant data [47]. A record is kept of the states the objects were in before the execution started and in the event of an exception all participating objects are returned to that state. No detailed information about the exceptions is needed [4, 84]. The user needs to be informed that backward error recovery has taken place so that he or she knows whether to repeat any actions possibly taken before the exception. The user will also need some kind of message as to the possible cause of the exception to avoid a repetition. Enabling backward recovery can prove to be an ideal solution in many instances but it places a load on the entire system both in terms of time, space and performance. Some systems may not be critical enough to need this option.

- *Retrying - Retry or Re-Enter*

  This option generally implemented with the user's involvement. It is similar to backward recovery except that backward recovery is system initiated. Whenever an exception occurs due to input provided by the user, perhaps due to the incorrect number of parameters (argument validation) or the wrong type (syntax), a retry option should be generated and the user given another opportunity to provide input [9, 58]. The retry option can also be in the form of an "undo" function. The retry message also serves the purpose of informing the user of the exception. The danger of this approach is that it can lead to infinite loops if the user continues to enter erroneous data [47]. Unambiguous exception messaging is of great importance here, preferably providing an example of the correct input required. The restarting point must be clear [13, 72] and before an attempt to re-enter is made a stable state must be established [47]. (This could be done through the use of backward exception recovery.)

Diagram 3.13 illustrates the mechanics of this model:

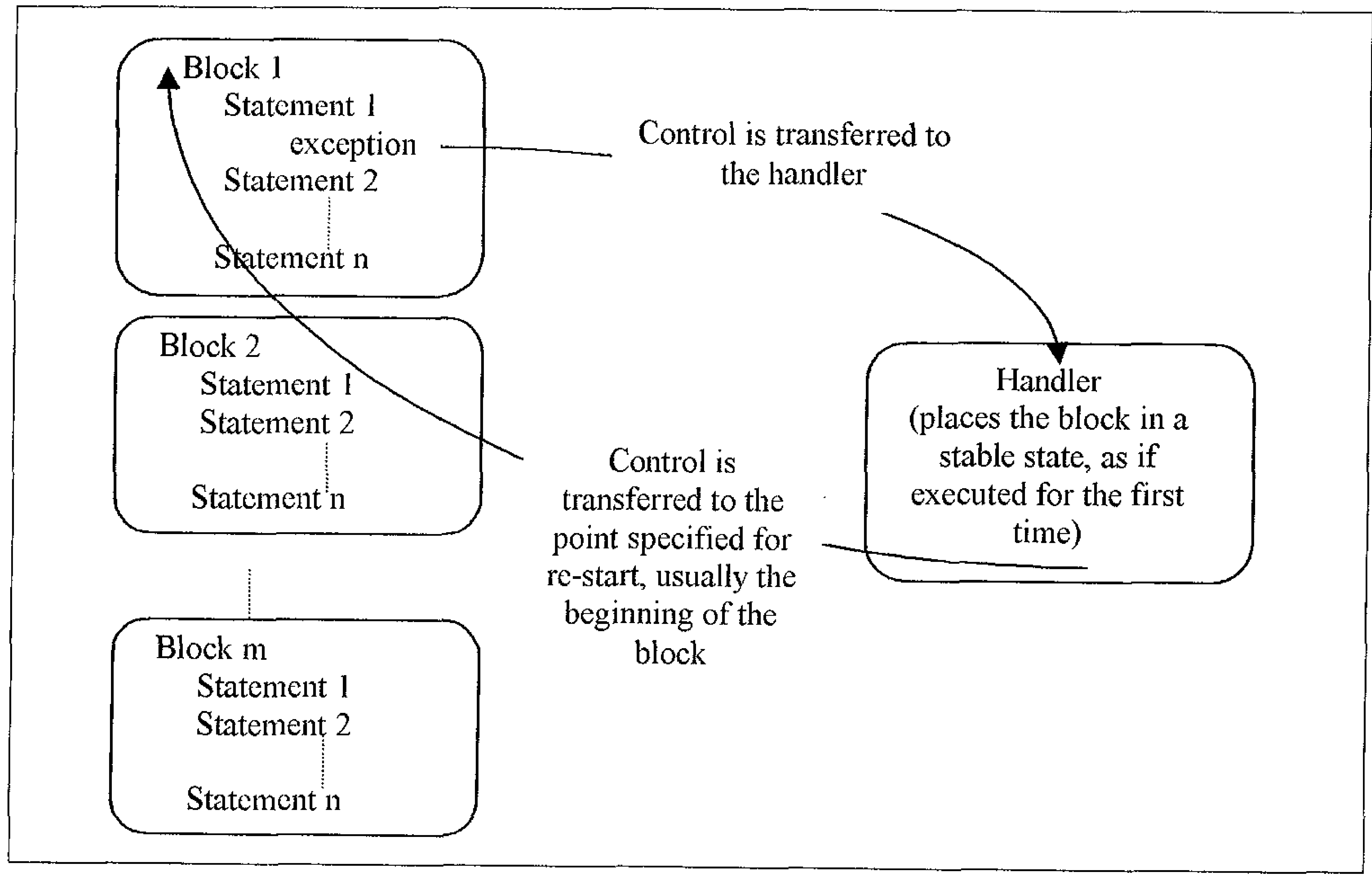



**Diagram 3.13 – Diagrammatical representation of the retrying model.**

This option can increase the flexibility and reusability of a system if a choice between models is provided depending on the circumstances [13, 27]. This can be done through dynamic handler allocation as mentioned in Section 3.4.3.1. Different handlers can be used to implement different models, or a hierarchical structure can be used typical to object-oriented development whereby the handlers for the different models inherit the basic functionality from a super class and add specific functionality.

Termination, retry or resumption may not always be possible. In this case the following options may be followed, listed here in order of preference:

- *Abort*

  If it is impossible to resume or retry, implying a failure of the system, without being "hung", controlled termination is the only option left [4, 9, 13, 40]. A stable state should be acquired [47] before termination, possibly through backward exception recovery. In this case the control is passed to the handler that completes the execution [13]. The control is never passed back to the calling routine. A message should be generated to

inform the user of the termination and what is to be expected on restart, i.e. inform the user whether a stable state was acquired (or not!).

- *Restart*

  An exception occurs when the control is never passed to the caller such that the system becomes "hung". The detection of a restart may vary, but in general it may be accepted that inactivity and non-reaction for a fixed period of time may be an indication that a restart is needed. The only solution then is for the user to restart the system, since it may not be possible for the system to restart itself [40]. It is difficult to generate a message since the flow of control in the system cannot be directed. The effects of this may be disastrous for the application – data might be lost - and the user needs to make his/her own conclusion as to what is happening.

- *Catastrophic*

  Machine crashes and needs to be rebooted [40]. Distributed systems will be likely to suffer from this "model". The complexity of concurrent processing in a distributed system makes it nearly unavoidable. The minimum requirement would be for all processes to release acquired and shared resources in the case of crash.

According to the three laws of exception handling of Mandrioli and Meyer [47] exception handling cannot be successful if these laws are not adhered to:

1. *The first law: A routine may only terminate by either fulfilling its contract or failing to fulfil its contract.* There is no middle ground or alternatives. Either a routine does what it promised or it fails to do so.
2. *The second law: The failure of a routine must always lead to an exception in the calling routine.* If a routine fails to fulfil its contract the calling routine must be notified otherwise the calling routine will continue and the exception will be ignored leading to inconsistencies and possibly complete failure.
3. *The third law: A routine may only react in two possible ways to an exception – return all the objects to a stable state and either attempt continued execution or terminate execution.* In the instance where an exception is raised the calling routine may only react in two ways – repair one way or the other, if possible, otherwise end execution. Again there is no middle ground.

## 3.6 Summary

This chapter forms an overview on the major different aspects of exception handling. It showed in Section 3.2 that there is a need for incorporating exception-handling policies into the software development life cycle from as early as the design phase, and in Section 3.3 it summarised the existing policies on exception handling. These policies are implemented through strategies, as described in Section 3.4, that cover the life cycle of the exception from detection and propagation to the handling of exceptions [4]. The final part of this chapter, Section 3.5 dealt with the different models for handling exceptions. The next chapter examines the major object-oriented implementation languages in order to establish how they make provision for the exception-handling models. Reporting exceptions and displaying exception messages will be dealt with in Chapter 5. Table 3.14 summarises the contents of this chapter:

| **The Exception Life Cycle** | | | | | |
|---|---|---|---|---|---|
| **Stage in an exception's life cycle** | | **Comment** | **Preparations in software development process** | **Problems** | **Chapter/ Section Reference:** |
| 1 | A fault occurs leading to an error causing a failure. | Exceptions cannot be eliminated – they must be dealt with. | High level polices are specified for dealing uniformly with exceptions. | Realisation of these policies. | Sections 3.1 - 3.3 |
| 2 | Detection of the failure as an exception. | To detect an exception it must be related to a concrete construct. | Use violation of constraints and contracts. | Exceptions exist which cannot be detected as violation constraints or contract breaches. | Section 3.4.1 |
| 3 | Identification and classification. | Use characteristics to identify exceptions. | Can be based on origin, cause or effect. | A single practical classification is difficult to construct. | Section 3.4.2 |
| 4 | Notification and reporting. | Externally or internally by means of messages, error codes or logs. | Responsibilities must be assigned. | Inconsistent assumptions might lead to missing or duplicate actions. | Section 3.4.3 (and Chapter 5) |
| 5a | Handling. (successful) | Exception resolution is provided in one form or another. | Terminate, retry or resume. | Ensuring stable resources and data structures. | Section 3.5 |
| 5b | Handling. (unsuccessful – failure) | No exception resolution is provided. | Restart, abort, or catastrophic. | Guarantees regarding the state of resources and data structures. | |

**Table 3.14 – Summary of the issues related to exceptions in the exception life cycle.**

# Chapter 4 – Object-Oriented Languages And Exception Handling

## 4.1 Introduction

In Chapter 3 the different exception-handling policies, strategies and handling models were examined in general. The exception-handling constructs of different object-oriented languages are described in this chapter in order to be able to establish whether the constructs allow for a realistic implementation of the models and to identify the possible shortcomings of the language specific exception-handling constructs.

*"It is no longer possible to consider exception-handling as a secondary issue in language design, or even worse, a mechanism to add after the fact via a library approach"* [13]. It is a primary feature of a programming language and should be integrated with the features of the language [13]. Mandrioli and Meyer [47] are of the opinion that the available programming language constructs give a false sense of security to programmers - these constructs are not as safe to use as the programmers would hope. These authors are also of the opinion that standard programming control structures should be used rather than exception-handling structures. One reason for this might be the difference in interpretation between the language developer and the language user as to exactly what facilities the language provides. Bail [4] echoes this with his statement *"Languages that provide exception-handling features generally include a set of predefined exceptions that cover specific erroneous conditions. Proper use of these predefined exceptions is essential to preserve their effectiveness, and to ensure that their use is consistent across multiple systems"*.

The emphasis should be on writing the code for normal execution, not on writing exception-handling code. The use of the exception-handling structures should be reasonably enforced but there should obviously be a balance between the code for normal execution and the code for exception handling [58].

The rest of the chapter is organised as follows: Section 4.2 motivates why the object-oriented paradigm is chosen and Section 4.3 investigates and compares the implementation details of exception-handling constructs in different languages. Section 4.4 concludes with final remarks.

## 4.2 Why the Object-Oriented Development Paradigm?

A paradigm provides a framework and thus structures the thinking process about a specific subject area. Object-oriented development is a popular paradigm and has proved itself to be practical for a wide variety of systems. It can be used not only for the development of sequential centralised systems but also for real-time, distributed and concurrent systems [13]. This is to the advantage of developers since the one paradigm can be applied for the development of different types of systems. It serves its purpose well to enable developers to manage the complexities of system design and maintenance [85].

The paradigm also allows for a reasonably straightforward method of modelling real-world situations in their facets as being entities (either concrete or abstract), having characteristics, and behaving in a specific manner. Objects are modular units used to encapsulate the facets of the real-world entities. The objects interact according to specified relationships as derived from real-world relationships. Relationships are also modelled with the object-oriented paradigm. It allows for the possibility of reuse, which is quite important since development is expensive and time-consuming.

Existing notations, such as UML (Unified Modeling Language), and modelling languages are well developed to support software development through the whole life cycle with this paradigm. The hierarchical class structure of the object-oriented development paradigm allows for the implementation of different degrees of specificity along a hierarchy, where the top part of the hierarchy represents a generalisation and the nodes lower down more specifics on the generalisation or abstraction [13].

Despite the widespread use of the object-oriented development paradigm, few object-oriented languages are equipped with exception-handling structures that really integrate it with the object model [26].

The following are general requirements for languages that provide built-in exception-handling constructs:

- Exception-handling constructs should be flexible enough to allow for different exception conditions and models.
- Uniform implementation should be allowed throughout, regardless of the type of exception [9].

- The language should enforce a controlled and disciplined implementation mechanism.
- The exception-handling constructs should be well structured.

Yemini and Berry mention an additional very important aspect of exception-handling constructs in languages [86] namely that the exception-handling constructs should be *orthogonal* to the language itself, meaning that the language should not be affected if the constructs are not used and that the construct should not be used for other programming tasks besides exception handling [8]. If this is not the case then the language cannot exist independently of the exception-handling constructs, thus creating dependencies that might lead to additional complications.

The following languages were researched: Ada, C++, Eiffel and Java. These languages are widely used for software development, and all provide mechanisms for implementing the object-oriented software development paradigm and comply with the requirements set out above.

For each of these languages the following will be discussed in the following section with code-segment examples. This issues covered are:

- Exception detection.
- Declaration and implementation of exceptions in the code.
- The change in flow of control between the point(s) where exceptions are raised and the handler(s) thereof.
- The underlying built-in method of exception handling.

## 4.3 Languages

In Sections 4.3.1 – 4.3.4 reference will be made to a "block of code" which implies multiple statements combined to form a single functional unit as defined in Section 3.5 [13]. A very simple example is used for the discussion of the exception-handling constructs in the code. It involves a value being entered by the user and the square root of the value calculated. The value entered is tested and if it is found that the value is less than zero an exception must be raised. These languages each contain a pre-programmed mathematical function for the calculation of the square root of a value. By definition, when working with real numbers, the square root can only be calculated of a value greater than or equal to zero. If an exception is not raised when a value

of less than zero is entered the routine will try to calculate the square root of the value, which is not allowed, and the routine will fail.

Each of the code sections implements the same example. The keywords are printed in black and the variables and non-keywords in blue. The sections dealing explicitly with exceptions and exception handling are printed in italics, purple and bold. Comments or explanatory notes are in green.

### 4.3.1 Ada [47, 79]

Exception handling in Ada is implemented by declaring an object of type *exception* in the *Package* specification. The name of the exception is given followed by a colon and the word *exception* [79]. (See code segment 4.1, 1) The implementation details are then given at the end of the specification. Exceptions' specifications are always written last, to separate the code from the rest of the normal execution code.

Ada implements exceptions as constraint violations with, in general, the use of the conditional *if-then-else* statements. In the example the constraint that must be tested for is that the value of the variable *x* is greater than or equal to zero. The *if*-part tests the value of *x*. (See code segment 4.1, 2).

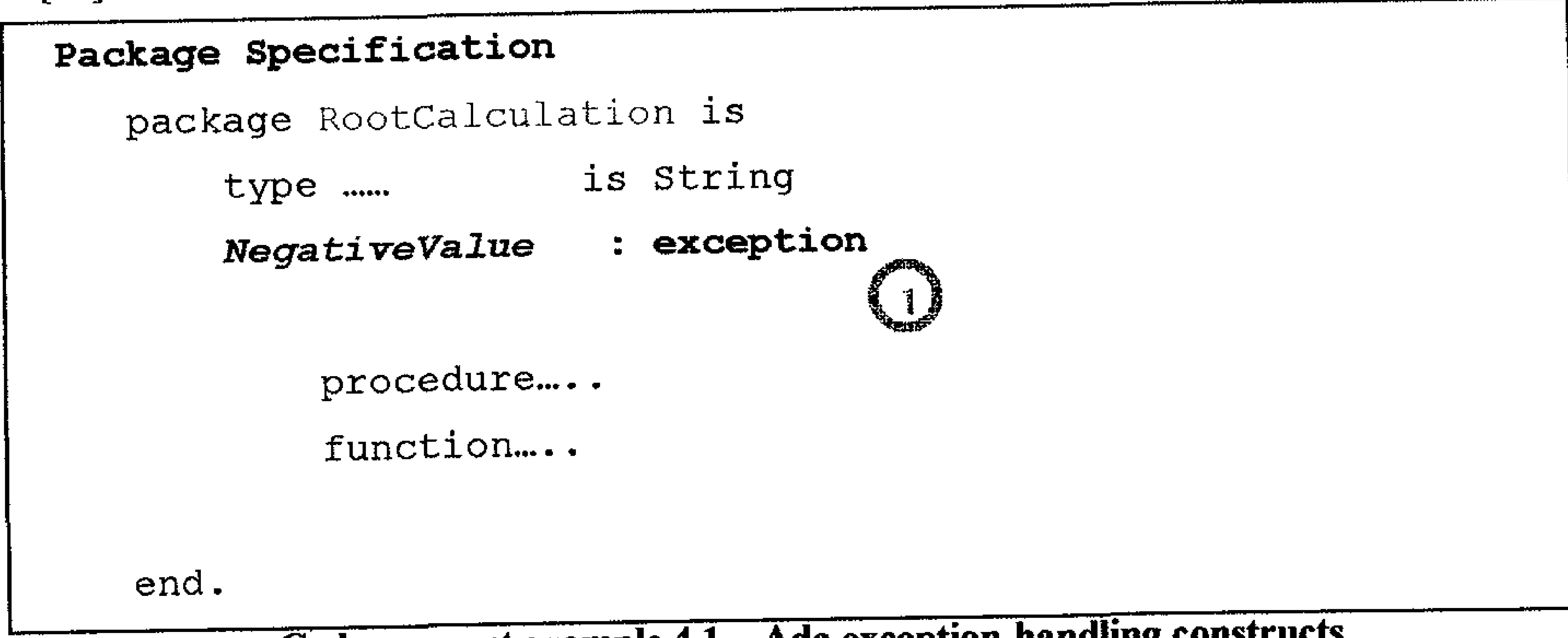
If a violation is detected, the *then*-part is executed, otherwise execution is continued with the *else*-part. The exception is raised by the keyword *raise* followed by the name of the exception. (See code segment 4.1, 3). The exception handler, identified by the name of the exception after the *raise* keyword, is then called, and the flow of control is transferred to the section starting with the keyword *exception* followed by the keyword *when* and the name matching the exception, in this case *NegativeValue* [79]. (See code segment 4.1, 4). The statements after the name of the exception indicate the actions to be taken in the case of the exception being raised.

The termination or resumption models can be implemented with the aid of the code following the *when* keyword [13, 79]. Termination can be implemented by, for example, printing a statement stating that the value is invalid. Control is not passed back to the routine. In the case of termination the block of code containing the exception is terminated and execution continued after the exception is handled with the first statement after the block.

Resumption can be implemented by taking corrective action and returning the flow of control to the routine for continuation. (See code segment 4.1, 7). Termination is implemented before the "OR" and resumption after the "OR"). The *return* keyword returns flow of control to the caller [47]. (See code segment 4.1, 5). Thus it signals the end of the exception correction actions. For all exceptions not explicitly prepared for, knowingly or unknowingly, a keyword *others* may follow the keyword *when*. All unexpected exceptions and exceptions not explicitly provided for will be propagated to this default handler [79]. (See code segment 4.1, 6).

Although the programmer can write his or her own code for exceptions, there are also built-in exception handlers for common exceptions such as division by zero. If a handler cannot be found which is local to the routine, the enclosing routines are searched and if none is found or the default handler at *when others* cannot handle the exception, the system default is executed - which is to abort execution [79].

The following statements by Thomas *et al.* [79] "Since constraint violations occur only infrequently (or at least that is what *should* happen)..." and "At the start of execution all exceptions are deemed not to have occurred..." indicate that certain assumptions are made regarding the occurrence and existence of exceptions when the languages are developed. The programmers should take these assumptions into account when designing a system in order to make sure that the assumptions will hold. Mandrioli and Meyer say *"Too often the Ada mechanism lures programmers into believing that by just raising exceptions they can forget about awkward cases. But in the end this only makes the system either unsafe or more complicated"* [47].



```
Package Specification
   package RootCalculation is
        type ......          is String
        NegativeValue    : exception
                                       (1)

            procedure.....
            function.....

   end.
```

**Code segment example 4.1 – Ada exception-handling constructs.**

**(continued on the next page)**

```
Implementation Details
with RootCalculation
use  RootCalculation
    procedure Root is
              x : Real;        y : Real;
      begin
              put("Enter Value:");
              get(x);                          (1)
           (2) if x < 0.0    then raise NegativeValue
                             else y:= sqrt(x);
                  end if;
     exception
        when NegativeValue (3)
              => put("Negative Value - Invalid");
              OR (4)
              => put("Enter value:");
              => get(x);
           (5)
return;
        when Others  (6)

      end RootCalculation;
```

**Code segment example 4.1 – Ada exception-handling constructs.**

**(continued from the previous page)**

### 4.3.2 C++ [45, 51]

An exception is declared as an instance of a class and referenced by its name. User-defined or built-in classes can be used. The *try* keyword is used to denote code that might lead to exceptions when executed. (See code segment 4.2, 1).

C++ 's exception-handling facility is used for responding to run-time program anomalies detected as violation constraints. The *if-then-else* statement can in general also be used in C++ to detect

violation constraints. The *if*-part of the statement tests the input value and if the test is passed, execution continues with the *else*-part. In the case of an exception the *then*-part is executed. (See code segment 4.2, 2). The *throw* statement executes when an exception is realised. It then transfers the flow of control to the exception handler with the matching name, in the *catch* block, if such a handler exists.

The exception handlers are specified in the block of code directly following the *catch* keyword and the exception's name. The *catch* keyword is followed by the names of the handlers matching those types of exceptions that can be called. The code that follows contains the instructions to be executed for the specific exception. In this example only a message is printed on the screen indicating that the value entered is invalid. (See code segment 4.2, 4). When a handler is successful, the *return*-keyword returns the flow of control to the routine's caller, at the point following the *try* block of the *catch* handler, not at the point following the *throw* statement [45]. This indicates that the termination model is used as foundation for the development of C++ exception-handling constructs [13].

Different handlers are listed after the *catch* keyword [45]. If a handler cannot be found within a *try* block the enclosing *try* blocks are searched until the system is terminated, by default, if no matching handler is found. (See code segment 4.2, 3).

C++ also separates the exception-handling code from the normal processing code but not as pertinently as is the case with Ada [45].

The use of the exception-handling constructs in C++ have also not been without dispute: *"Like any other feature of C++, exceptions can be used to simplify the implementation process and make programs more robust and easier to maintain, or they can be misused, adding yet another degree of complication to programs making them a maintenance and testing nightmare"* [50].

```
void  RootCalculation()
{
   real  x, y;

   puts("Enter Value:");
   gets(x);
      try (1)
      {                  (2)
            if  x <0 .0  throw NegativeValue
                         else y = sqrt(x);
      }
                         (3)
      catch NegativeValue
      {
            puts("Negative Value - Invalid") (4)
            return; (5)
      }
}
```

**Code segment example 4.2 – C++ exception-handling constructs.**

### 4.3.3 Eiffel [69]

An exception is defined by a name (string). Eiffel formalises the violation constraints with the implementation of the idea of a contract between the calling routine (caller) and another requiring a service (callee). Whenever the callee cannot fulfil a contract the caller must be notified. The code for exception handling is separated locally from the normal execution code, within the routine.

The precondition(s) must be fulfilled before execution of the routine body (See code segment 4.3, 2) can commence. Before the square root can be calculated it must be ensured that the value of *x* is greater than or equal to zero. (See code segment 4.3, 1). After the routine body is executed

the post condition must be ensured, in this example the value of *y* must be greater than or equal to zero. (See code segment 4.3, 3).

```
RootCalculation (arguments   :  TYPES)  is
   require    pre_condition  (1)
         x > = 0.0

   local
         x : real;
         y : real;

   do          routine_body
         y := sqrt(x);           (2)

   ensure            post_condition    (3)
         y > = 0.0

   rescue (4)
         if  x <0.0 then get_new_x_value
         x := 1; }    Restores Invariant  (5)
         y := 1; }

  retry  (6)

end.
```

**Code segment example 4.3 – Eiffel exception-handling constructs.**

If either of these constraints is violated the flow of control is redirected to the *rescue_clause*. (See code segment 4.3, 4). Eiffel allows for two broad approaches to exceptions – the one is to create a stable state and terminate (organised panic) [69], the other to change the conditions and retry [13, 69]. The handler, denoted by the *rescue_clause* is executed when an exception occurs and assures the stable state before termination or retry. It always restores the class invariant (See code segment 4.3, 5) thus it voids the effect of the exception to create an "all-or-nothing" effect. In

the case of termination the *rescue_clause* completes, the routine terminates and the routine's caller is notified.

It may also contain an optional *retry* option. (See code segment 4.3, 6) The code in the *rescue* option can provide for a correction of the current problem and the *retry* option will usually re-execute the routine from the beginning. If the *retry* option is successfully completed no indication of the problem will be given to the caller. If a handler signals termination to it's caller, the process is repeated in the caller and its caller until the system default is executed - termination with a message describing the cause [69]. A routine without a *rescue_clause* has an implicit *default_rescue* clause with null body.

The following statements give an idea of the Eiffel language's philosophy regarding exceptions: *"...exceptions should be used sparingly. They are not a technique for dealing with uncommon (but acceptable) cases. They should be reserved for unpredictable events, unstable preconditions, and protection against exceptions remaining in software"* [69].

### 4.3.4 Java [28, 51, 57, 82]

Exception-handling code in Java is embedded within the normal execution code, but conceptually it is separate from the normal execution code. Exceptions are declared as instances of classes and referenced by name. User-defined or built-in classes can be used. Exceptions detected as values are tested by means of the *If-then-else* statement for violation constraints.

All the code that might throw an exception should be put in *try* blocks. (See code segment 4.4, 1). The implementation details for the handlers are contained in the correspondingly named *catch* blocks. When an exception occurs, the flow of control is passed from the *try* block to find the matching *catch* handler. In the case of an exception no code after the *throw* statement is executed (See code segment 4.4, 2), but the code following the *try* statement is executed, in other words, the code starting with the keyword *catch* [48]. (See code segment 4.4, 3). The *return* keyword forces the current routine to exit immediately and returns the flow of control to the calling routine. (See code segment 4.4, 5). If there is a part of the code that must be executed at all costs, the *finally* keyword can be used, for instance where a clean-up operation is needed. (See code segment 4.4, 4). For re-raising an exception, the keyword *throw* in a *catch clause* can be used. This is useful when the programmer wants to process a part of the exception him/herself and then return it to the Java handlers for further handling.

```
class  MathCalculations
  {
  void RootCalculation() throws NegativeValue

  void  RootCalculation()
  {
     real  x, y;

     puts("Enter Value:");
     gets(x);
        try (1)
        {                    (2)
             if  x <0 .0  throw NegativeValue
                          else y = sqrt(x);
        }
           (3)
        catch (NegativeValue)
        {
             puts("Negative Value - Invalid");
             finally (4)
             return; (5)
        }
  }
```

**Code segment example 4.4 – Java exception-handling constructs.**

Terminate semantics form the foundation of this language - continuing with the first statement after the *try* statement [13, 51]. Where no appropriate exception-handler can be found the exception is passed on to the caller objects until the system aborts with an exception message.

Manning says, *"Sometimes, even in debugged and tested applications, things can go wrong that are beyond the control of the program. Problems can be caused by the environment in which your program is being run, or the user will do something your program didn't anticipate.*

*Whatever the problem, a well-designed application should be able either to handle the unexpected or at least exit gracefully"* [48].

### 4.3.5 Comparison of Object-Oriented Languages and Exception-Handling Capabilities.

Table 4.5 compares the different languages with respect to the important issues surrounding exception-handling construct functionality provided in languages.

| | ADA | C++ | EIFFEL | JAVA |
|---|---|---|---|---|
| **Comparison with respect to specific language characteristics:** | | | | |
| **Separates exception-handling code from normal execution code:** | Separated through objects called exceptions. | The code is contained in a separate block of code denoted by keywords. | The code is contained in a separate block of code denoted by keywords. | The code is contained in a separate block of code denoted by keywords. |
| **Exceptions implemented as [26, 51, 85] either:** • **data objects** • **full objects** • **strings (names)** | Full objects | Data objects (instance of a class) [26] | Names (numbers) [26] | Data objects (instance of a class) [26] |
| **Keywords:** **Declaration:** | e_name : *exception* | | | |
| **Detected:** | *if...* | *try...* *if...* | *rescue* *if...* | *try...* *if...* |
| **Evoked / Raised:** **(Exception handler is called)** | *raise* e_name | *throw* e_name | | *throw* e_name |
| **Implementation (Handler):** **(Exception handler code)** | *exception* *when* e_name *return* | *catch* e_name | *retry* | *catch* e_name |
| **Unexpected Exceptions:** | *when* others | | | |
| **Clean-up Operations:** | | | | *finally* |
| **Granularity, handlers can be attached to [13, 51, 85]:** • **statements** • **blocks/routines** • **methods** | Block level [13] (Extended Ada) Methods, Objects or Classes [85] | Statements [85] Block level [13, 51] (Extended C++) Methods, Objects or Classes [85] | Block level [51] Methods, Objects or Classes [85] | Block level [51] |

| • **objects**<br>• **classes** | | | | |
|---|---|---|---|---|
| **Orthogonality [86]:** | Most orthogonal, it will be possible to produce code quite easily that does not make use of any exception-handing construct due to the fact that exceptions are declared as separate objects and the handlers are specified separately from the rest of the code. | In the case of C++ this will be more difficult, the *try-catch* relationship is interleaved with the language far more than the *objects* of Ada. | Least orthogonal, Eiffel is the most dependent on the exception structure and will certainly be most difficult to produce code without using the built-in exception-handling constructs. | In the case of Java this will be more difficult. The *try-catch* relationship is interleaved with the language far more than the *objects* of Ada. |
| **Comparison with respect to general exception-handling issues:** | | | | |
| **General Policy:**<br>**Reference: Section 3.3** | React-Repair | React-Repair | React-Repair retry | React-Repair |
| **Detection:**<br>**Reference: Section 3.4** | Violation of constraints by means of conditional constructs<br>*if-then-else* | Violation of constraints by means of conditional constructs<br>*if-then-else* | Contract Theory in collaboration with violation of constraints by means of conditional constructs<br>*if-then-else* | Violation of constraints by means of conditional constructs<br>*if-then-else* |
| **Flow of Control and Propagation:**<br>**Reference: Section 3.4.3.1** | Flow of control is passed to the handler with the keyword *raise*. The control is passed back to the routine with the keyword *return*. | Flow of control is passed to the handler with the keyword *throw*. | Flow of control is passed to the handler with the keyword *rescue*. | Flow of control is passed to the handler with the keyword *throw*.. |
| | If no handler is found, the exception is propagated until a default handler is executed. | | | |
| **Responsibility:**<br>**Reference: Section 3.4.3.2**<br>• **Detection**<br>• **Notification**<br>• **Handling** | The detection and notification goes along with violation of constraints and is the responsibility of the routine that will detect the constraint violation. The responsibility for handling lies with the handler and ultimately the default handler, if none other can be found. | | | |
| **Handling Model:**<br>**Reference: Section 3.5** | Termination [47, 86]<br>Resumption [47] | Termination [51] | Termination<br>Retry [51] | Termination [51] |

**Table 4.5 – Comparison of object-oriented languages.**

These languages seem to be reasonably equal in the functionality they provide and there is not a definite indication that the one is more adequate than the other. There are more references and examples in the literature to Java and Ada than to the other two languages, and this could be taken as an indication that they are the preferred languages. All of these languages lack true flexibility in that they do not provide for different exception-handling models. It might not be impossible for a programmer to manually implement a model other than the default provided but this would violate the language-inherent model. This could lead to other problems since the language would not used as intended.

## 4.4 Summary

From Sections 4.2 and 4.3 it seems that situations should never arise where a system is restarted after being "hung". The implementation seems to be simple and straightforward. The question begs to be asked: Must attention still be given to exception handling during the other phases of the software development life cycle, besides the coding phase, if it can be implemented so easily – and provision made for unexpected events with default handlers? Reality gives the answer to this question. It is obvious that exceptions happen quite often, and although care is taken and provision made for exceptions, programs still "crash". One reason might be the *"Lack of experience with these features could lead to ineffective use and even to degradation of program behavior"* [4]. Another reason might be the poor documentation on exception handling in programming manuals. It is often added in an appendix with only a few general examples, which are usually not detailed, and the vague explanations leave many unanswered questions.

The conclusion is that the implementation of exception handling in later phases of the software development process is not completely successful. Successful implementation does not only rely on the correct coding but also the *correct thinking* about the exception in relation to the whole system. To solve the problem certain aspects concerning exceptions must be addressed during earlier phases to ensure that they are adequately provided for. Cargill refers to exceptions in C++ *"The popular belief is that exceptions provide a straightforward mechanism for adding reliable error handling to our programs. On the contrary, I see exceptions as a mechanism that may cause more ills than cures. Without extraordinary care, the addition of exceptions to most software is likely to diminish overall reliability and impede the software development process."*

This chapter investigated the (internal) implementation details of exception handling, Chapter 5 will investigate the matter of reporting exceptions visibly to the user (externally) and will also motivate the necessity of reporting exceptions.

# CHAPTER 5 – REPORTING EXCEPTIONS

## 5.1 Introduction

The quality of software depends not only on the detection, propagation and handling of exceptions but also on the adequacy of the messages generated and displayed when an exception occurs. The way in which the message is reported can have various effects on the user, the system and ultimately the environment. Human operators can be seen as components of a computerised system, and even more so as *critical* components in safety critical systems. Communication or interaction failures between the human and the computer system can have serious consequences, just as with any other component, in such a system. [24]. Successful communication - understandable communication - influences as a consequence, the usability of the system. Usability is related to the reliability of the system and as usability improves, so does reliability and as a consequence the quality and of the system [68].

The effects of the exception can be either reduced or exacerbated depending on the effectiveness of the communication with the user. If the user is expected and able to perform corrective measures timeously the effects will be less in the case of clear communication than in the case where the user is unable to establish what is required of him or her. The actions of the user can have an overall influence on the dependability and reliability of the system [24].

The rest of this chapter will discuss the reporting of exceptions to users. Section 5.2 motivates why it is necessary to report exceptions to the user and Section 5.3 specifies the information that must be included in the message reporting the exception. Section 5.4 investigates the presentation of the messages to the user and Section 5.5 sets out some of the established requirements for successful reporting of exceptions. Section 5.6 introduces new developments in exception reporting while Section 5.7 summarises the chapter.

## 5.2 The Necessity of Reporting Exceptions

Communication between the user and the system is essential for the successful and useful operation of a software system. Computer systems must support the user's tasks, not only in normal processing but also, and even more so, during exceptional processing [66]. During

exceptional processing the system can support the user by communicating with him/her through messages. Communication becomes even more important when an exception occurs as it facilitates the task of recovery [66]. The user is accustomed to the normal operation of the system and knows what to expect and what to do during normal operation. The user is less accustomed to the procedure in the case of an exception and information needs to be communicated to the user in such circumstances. The user perceives the system as friendly or unfriendly depending on how an exception is handled, rather than on whether the system is functioning correctly. The user will find an informative message friendlier and more useful than a message that states "unknown application error" [11]. The result of the latter might well be an infuriated user. Users do not necessarily expect systems to function perfectly without any occurrence of exceptions since exceptions have always been part of their experiences with software systems. They do, however, expect to be informed about exceptions in an understandable way. They also need to be told whether the effects of the exception on the system and environment can be controlled. This gives the user a means to assess the damage caused by the exception [53].

It is necessary for the user to know how to proceed [66] since there is no "correct" model to follow in the case of an exceptional condition. A message is used to inform the user of the current state and to prompt him/her to respond according to what is required. It promotes understanding and lessens the effects of the exception. The next section will describe the information that must be included to realise successful exception reporting.

## 5.3 Information to Include when Reporting Exceptions

As mentioned in Section 5.1 effective communication can limit the extent of the damage of an exception. Maxion and Olszewski [49] list the following criteria to be used to evaluate the adequacy of messages, ranging from the least to the most strict:

- No message is displayed.
- A message is recorded internally.
- A message is displayed but the exception is misidentified.
- A vague or unspecific message is displayed.
- A message is displayed and the exception correctly identified.
- A message is displayed, the exception correctly identified and located.

An adequate message thus conveys information, firstly about the fact that an exception has occurred and, secondly, the type of exception that has occurred. Besides that, it should not

convey irrelevant information [65]. The specification of the location can be enhanced with additional information as to the cause, resolution and possible effects of the exception.

The displayed information influences the usability of the system. Lewis *et al.* [44] commented that poor or inadequate knowledge would lead to poor problem solving due to poor understanding. Poor understanding leads to difficulty in learning the system. Exceptions must often be reported to the user and information regarding the exception displayed [4]. The user has to be informed that an exception has occurred, that normal processing is no longer an option and that a resolving strategy must be implemented in order to continue normal processing. The user has an idea, a mental model, of what is happening during normal processing but when an exception occurs the user's mental model will not be updated unless he/she is informed or obliged to do so [67]. This model forms the framework for the user of the interaction process, and the effectiveness of the communication will determine how accurate the framework is at any given time [12]. The user's mental model must be updated so that it is clear

1. what has happened before the exception,
2. what the current processing status is,
3. what must happen to resolve the exception, and
4. what is likely to happen after the resolution of the exception.

The update process is especially important since the user will not have had reason to remember what he or she had been doing before the exception to cause the exception [67]. Due to the exception the usual course of execution, to which the user is accustomed and which is expected, has changed, and the user might feel disoriented and not know what to expect and how to continue. If the user is required to take additional steps in order to resolve the exception, clear instructions should be given as to the effects.

There is a difference between system detection of an exception and user detection of an exception [67]. The system detects an exception due to violation of constraints and must translate this violation for reporting purposes into a form that is understandable to the user in terms of concepts with which he/she is familiar. Before a decision can be made as to *what* information must be reported a decision must first be made for *whom* the message is intended. An application programmer has need of detailed technical information to assist in the debugging process while the messages reported to end-users must be translated to relate to the goals and tasks of the user

on a higher level [66]. It is very difficult to accomplish the successful reporting of exceptions to a variety of users.

Table 5.1 provides a summary of the information that must be reported to the user in order for a message to meet certain minimum requirements of adequacy and this is further illustrated by means of examples in Figures 5.2 and 5.3. Adequacy can be defined to be *the length of the time taken by an operator to locate a problem when presented with the given message* [49]. Maxion and Olszewski recommend the following when reporting an exception. However, it is debatable whether the end-user really needs to know the name or high-level description of the exception. It seems as if a strong case could be made for exception reporting based on the role of the user. Thus an end-user and a programmer would be given different types of reports, since the programmer is obviously very interested in details like the name of the exception and the line number in the program where the exception occurred.

| Information | Description | Example |
|---|---|---|
| **Name** | The name of the exception must be displayed in order to identify the exception and differentiate it from other exceptions. | "Exception!" or "Authentication Exception" |
| **Description** | The exception must be described as precisely as possible. | "Access denied due to invalid identification" |
| **Location** | The place of origin (and for the developer even a line number) must be indicated. | "Remote server denied access." |
| **Cause** | All the possible causes of the exception must be listed. | "Invalid username or password." |
| **Resolution** | The possible strategies if more than one is available, and what is required to implement the strategy. | "Retype the username and password." |
| **Effects on the user and the system** | The current- and expected state of the system. | "Request to read file denied. Retry?" |
| **Effects on the environment** | The current - and expected state of the user and environment. | "Operation terminated." |

**Table 5.1 – The information that must be included in a message.**

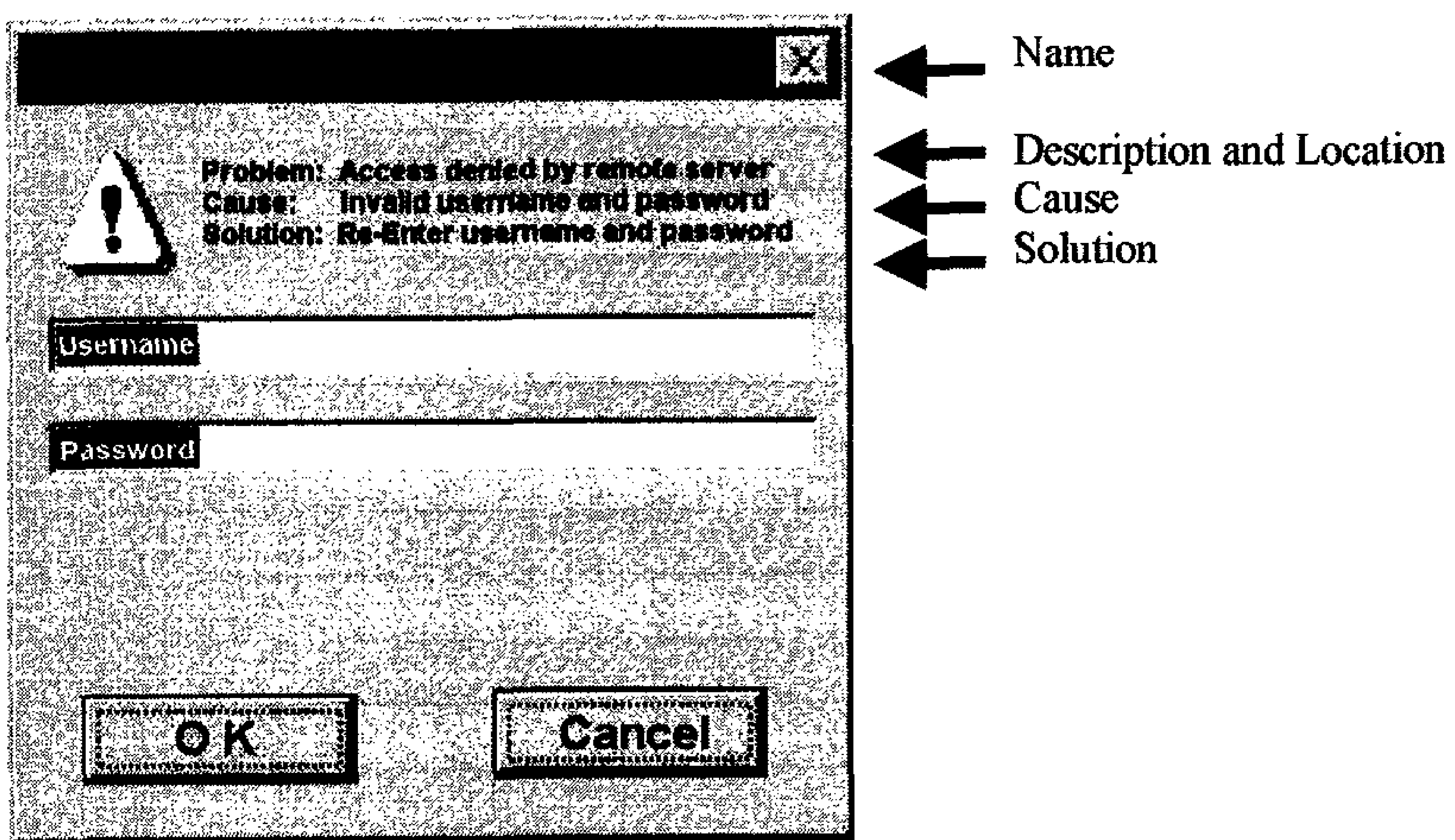



**Figure 5.2 - Example of information included in a message.**

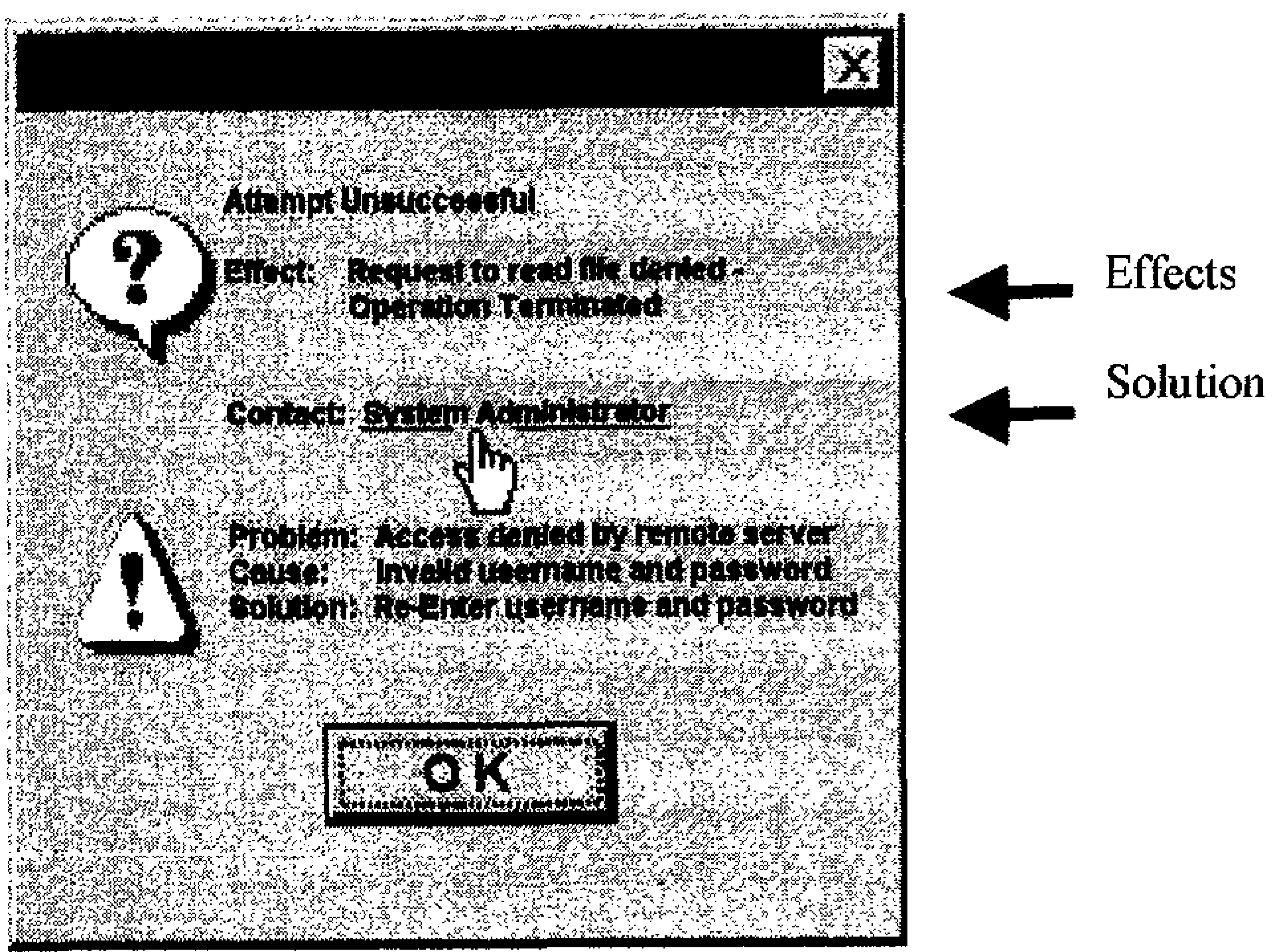



**Figure 5.3 -Example of information included in a message.**

The language must report the exception from the user's perspective and not from the perspective of the developer [67]. From the information presented the user must be able to complete a mental picture as to cause and effect.

There are trade-offs [4, 75] involved in the granularity, or detail, of the information displayed. These are listed Table 5.4 with indications as to the influence of the specific granularity trade-off on the adequacy of the message.

| Granularity Trade-Offs: Information Contained in Messages when Reporting Exceptions | |
|---|---|
| **Improves the adequacy of messages.** | **Does not improve the adequacy of messages.** |
| **Fine Granularity: More Information.** | **Coarse Granularity: Less Information.** |
| Recovery process and efficiency improves because more information is available for resolution. | Recovery process can be difficult and inefficient due to lack of information but may be suited to a more experienced user. |
| Increased complexity in design and coding due to the information that must be interpreted. | Less complexity, easier to implement. |
| Memory and processing requirements increase – overhead increases. | Less overhead required and better performance. |
| User's mental model improves due to detailed information displayed. | User may become frustrated due to lack of information provided. |
| Reuse of the same code for different exceptions not possible due to the specific descriptions. | The same message can be reused for related exceptions since the information contained is generic. |

**Table 5.4 – Granularity trade-off: Information contained in messages when reporting exceptions.**

This section dealt with the information that should be displayed when reporting an exception. The adequacy of the message depends on the granularity of the information but also "how" the information is presented to the user. If the necessary information is not presented in a correct manner it may be of little or no use. The next section will consider the presentation of information in more detail.

## 5.4 Presentation of Information

The information provided when an exception is reported is of little use if it is not reported in such a way as to effectively draw the user's attention to the message. The occurrence of an exception

is always unanticipated and no assumptions should be made, even for the most obvious cases of exceptions, about any foreknowledge the user might have about the exception. A clear message should be given, even in the case of an exception which is expected to occur frequently. The user must be able to understand, from the content of the message, the cause of the exception and what the proposed resolution strategy entails [67]. This is different from merely displaying information – the same information can be displayed in many different ways, but the level of understanding may differ for each. Effective reporting enhances understanding. As an example, the following messages could be used to inform a user that too many data items have been supplied (the "#" represents an integer value):

- "Incorrect number of values"
- "Too many values"
- "# values have been entered"
- "# values have been requested"
- "# values have been requested, # values have been entered"
- "# values have been entered too many"
- "Only # values allowed"

The above is not an exhaustive list. It is clear from the list that the same information can be conveyed in many different ways, but that there are definite differences in the clarity and descriptiveness of the messages.

The *purpose* of the message [4] should form the core of the reporting process, since it will influence the manner in which the message is compiled. It must correspond to the efficiency of the processing required, especially where the user is required to aid in the resolution process. In this case precise, detailed and unambiguous instructions are necessary. The following sections deal with some important issues concerning the reporting of exceptions. Section 5.4.1 explores general issues, Section 5.4.2 language related issues and Section 5.4.3 interface related issues.

### 5.4.1 General Issues

The following general issues should be taken into consideration when exceptions are reported:

- *Provide "Undo"* [44] *and "Cancel" functionality.*

  Involuntary movement, such as clicking on the wrong option, can cause exceptions. The users will, in many cases, realise the mistake and a quick "undo" function is mandatory in such situations. Extensive exception-handling constructs will waste time and effort.

Different levels of undo might be provided depending on the circumstances and the application. An undo function serves another purpose – that of preserving [56]. The user should preferably not have to repeat too many actions as more exceptions could be raised in the process. A "cancel" function might be more useful for operations occupying processor time that might take a long time to complete. Instead of the user having to wait for the operation to complete and then to undo, the operation could be stopped in the midst of processing to save time and the effects of the incomplete operation voided.

- *The occurrence of the exception must be reported as quickly as possible* [68, 86].
  It is conceptually much easier to establish a cause-and-effect, and so an understanding of the problem if it is reported when it is still evident. It also influences the amount of effort that will be required to resolve the exception condition.
- *No assumptions about any foreknowledge must be made.*
  When designing the exception description the designer must keep in mind that the user probably will not know what caused the exception. Clear and complete information must be provided so as not to leave any misinterpretation possibilities.
- *Do not over or under specify* [11].
  Over-specification may only confuse the user while supplying unnecessary information and under-specification might lead to ambiguities as to the cause of the exception and the effects thereof. Precise and to-the-point representation is necessary [56], which is difficult to accomplish due to the differences in levels of understanding of users - what one user will consider an under-specification the next user might judge as over-specification.
- *Provide context for the exception in relation to the processing.*
  Context should be provided in two forms, immediate feedback and archival feedback [66]. Immediate feedback refers to information about the current state of the system and archival feedback serves to give reference as to what preceded the exception. Context or history should be provided in reporting an exception to facilitate recognition instead of recall [68]. If the exception was caused by a series of actions the whole context should be provided. The user doesn't have to rely on memory to try and "remember" the context. If the last action precipitated an exception due to concatenation with an earlier action it must explicitly be mentioned so that the relationship between the actions and their outcomes is obvious [67]. A record of actions can be kept for this purpose in the form of an activity log [67] to be used in the case of an exception. Also a repertory of available resolution strategies can be implemented and the probable reaction of the system to these

can be provided [44]. Although this is ideal and very helpful to the user it increases processing overhead. It is also very difficult to implement. The actions taken by the system must be described to the user with great care. A *translation process* is needed and if it is taken into account that any number of actions can lead to an exception the possible combinations of actions are immense and a translation process very complex.

- *Help Functions.*

  Exceptions should provide cross-references to help functions using the same terminology; especially where it is not possible for detailed descriptions to be provided in the messages themselves. Help functions can serve the purpose of supplying more information about the exceptions if required [67]. If error codes are used, or exceptions are indicated by means of a numbering system, an option should also be provided for the codes and their descriptions to be printed, or it should be provided in printed form, or explanations of codes should be available as online help. Although help functions must be provided as an additional aid they should not free the exception-reporting mechanisms from responsibility of effective and informative reporting. A user usually turns to the help function as a last resort and having to use a help function is generally not perceived positively.

The issues mentioned in this section are general issues whilst the issues mentioned in the next two sections deal with more specific issues surrounding the physical presentation of information.

### 5.4.2 Natural Language-Centred Issues

The language used to report the exception should be on the level of the anticipated end-user although care must be taken when assumptions are made as to the level of users, as different users of the same system may have different levels of understanding [67]. The following are important:

- *Avoid the use of technical terminology and complex language* [67].

  The problem should be described in terms familiar to the user so as to facilitate understanding [56, 66]. Exception reporting with the aid of numerical error codes is a good example of poor readable reporting.

- *The use of correct and precise terminology.*

  Different terms can have different meanings for different users. Using clear and precise terminology minimises misinterpretation of terms used in reporting.

- *The tone of the message should not be accusing* [56].

  A polite, informative message will increase the usability of the system, as the user will perceive it as friendlier than an accusing message stating in no uncertain terms what the user did wrong [32]. Blame assigning should be avoided, although the user might be at fault, assigning blame will cause an emotional rather than rational response.

### 5.4.3 Human Computer Interface-Centred Issues

Language is not the only factor influencing the quality of the message. The way the message is brought to the attention of the user (visualisation), and the manner in which the contents of the message are conveyed to the user (structure), also plays an important role. The following sections consider issues that must be kept in mind regarding utilisation of the interface [67].

#### 5.4.3.1 Visualisation Issues

- *Attract the user's attention.*

  It might be that the message is displayed on the screen and the user's attention is elsewhere so that it might go unnoticed for a period of time [67]. More than one type of indication method can be used, perhaps a "beep" sound or flashing on the screen [66]. No assumptions should be made to indicate exceptions implicitly by means of, for example, the use of certain colours since different meanings are attached to different colours by different cultures [9]. Graphics should be used where possible instead of text – faster identification is possible through visual clues than by reading text that looks the same and visual clues facilitate recognition.
- *Differentiate unambiguously between exception reporting and normal processing.*

  Exception reporting should be done in a distinctively different and noticeable way from normal processing so as to emphasise and draw attention to it. Normal processing may also make use of reporting strategies to keep the user informed or for confirmation purposes, thus the distinction is very important since not all feedback or reporting involves exceptions [66]. Normal processing should not be allowed to continue *unless* some indication is given that the occurrence of an exception was recognised by, for example, clicking on a button - "OK". Thus, visibility and noticeability is of importance [56].
- *Uniform and consistent presentation of exception reporting.*

Message structure and format should be applied consistently and uniformly throughout the system to avoid confusion and to facilitate recognition. This implies consistently using the same features in the same proportions and positions for reporting throughout the system [56].

#### 5.4.3.2 Structural Issues

- *Allow for customisation.*

  Although not always practical, in systems where some exceptions are expected to occur frequently it might be advisable to provide customisation of levels of detail of reporting. As users become accustomed to the system they could be allowed to set up the option of displaying one-line exception messages instead of long descriptions [12, 67].

- *Provide examples or choices.*

  If input is required it can be useful to provide an example of the format of the desired input. It will allow users to compare [67] their input with the correct example and to draw conclusions. If possible, alternatives should be provided from which the user can choose but as few choices as possible should be allowed [44]. The provision of examples and choices reduces the guesswork, and the accompanying uncertainties, on the part of the user [56].

The issues mentioned in Sections 5.4.1 – 5.4.3 influence the degree to which the reporting of the exception will be successful. In order for the reporting process to be successful the message must be understood to such a degree that it can be acted upon, resulting in the resolution of the exception. The central issue here being the ability to understand and thus enhance the use of the system since it influences the quality of the software. Section 5.1 has outlined the correlation between understandability, usability, reliability and quality.

Testing software to uncover resilient exceptions is a recognised way of enhancing the quality of the software. In the same way reporting of exceptions to the user should also be subjected to testing in order to establish the functionality thereof. This can be done with usability testing by means of which the end-user is interviewed, surveys conducted, questionnaires completed or even observations made in the form of video recordings of the interaction process as perceived from the user's perspective [32]. The testing of usability will not be discussed here although some of the principles of supporting usability will be listed in Section 5.5.

## 5.5 Principles to Support Usability

Dix *et al.* [19] describes principles to support usability for an interactive system. The degree of usability is measured by users' experience with a system [19]. There are two types of "usabilities" that must be distinguished here: the usability of a system under normal conditions, and the usability of a system under abnormal or exceptional conditions. A system might attain a high usability score in normal operational conditions but a very low one under exceptional operational conditions due to, for example, lack of effective reporting. Usability in the latter instance is directly influenced by the communicating (reporting) of exceptions to users [67] as for successful interaction the user must have an accurate view on the state of the system and "keeping the user informed" relies on communication. The user's view must be consistent with the reality of the process to ensure successful interaction.

The implementation of these principles does not give any guarantees [65] that if they are blindly implemented the exception messages will be adequate. The perception of most developers is that it is difficult and complex to understand and implement these principles. "*Many authors have published guidelines in an attempt to assist application developers in providing this feedback to the end-user. In spite of this, adequate provision of feedback proves to be elusive*" [66].

Usability is not only important for the quality of the product but has further implications. Difficulty in using the system might lead to the software becoming obsolete. For example, an e-commerce web site with a high degree of usability will attract more prospective buyers than a web site difficult to navigate and use. The company whose web site suffers from poor visibility will not experience the same level of popularity and economic wellbeing as its counterparts despite the incentives it might offer. The web offers alternatives unlike software bought for personal or business use. When a user buys software for personal or business use he or she has a vested interest in finding out how it works and in dealing with exceptions. In the first instance the user is free to choose between many alternatives which are readily available, while in the second instance alternatives might not be a likely option. However, web sites reporting frequent exceptions will probably be abandoned forever. Users have many alternatives and do not need to be content with sites that behave poorly.

The effects of exceptions on systems are far reaching. They affect the immediate user and system but in the long term the environment and the consequences for the environment could be, as also

mentioned in Section 2.5, the most severe and damaging. The importance of effective exception handling will increase due to the possible consequences it might have. With reference to the example in the previous paragraph, developers will have to give more attention to detail regarding exception handling so as to improve usability. Improved usability can prove to be an important factor in gaining favour with users in a very competitive market where alternatives are readily available and unusable products may become obsolete.

The principles from Dix *et al.* [19] listed below can also be implemented when designing the structure and contents of messages to ensure maximum performance. The three main principles with their subcategories are: Learnability, Flexibility and Robustness and are reproduced in Table 5.5.

<table>
<tr><th>Learnability<br>defined as:<br>The ease with which users new to the system will be able to understand interaction.</th><th>Flexibility<br>defined as:<br>The different ways in which interaction can take place.</th><th>Robustness<br>defined as:<br>The level of support provided to the user to determine if a goal was achieved successful.</th></tr>
<tr><td>Predictability<br>The user must be able to predict the effect of an interaction based on previous experience.</td><td>Dialog initiative<br>Allow user freedom from artificial constraints on the input dialog.</td><td>Observability<br>Ability of the user to evaluate the internal state of the system from its perceivable representation.</td></tr>
<tr><td>Synthesizability<br>The user must be able to establish what effects previous interactions have had on the system.</td><td>Multi-threading<br>Support user interaction pertaining to one or more task at a time.</td><td>Recoverability<br>Ability of the user to take corrective action once an error is established.</td></tr>
<tr><td>Familiarity<br>The extent to which existing knowledge can be applied to the current system.</td><td>Task migratability<br>Pass control for execution to that internalised by user or system or shared between them.</td><td>Responsiveness<br>How the user perceives the rate of communication with the system.</td></tr>
<tr><td>Generalisablity<br>Generalisations within and across applications.</td><td>Substitutivity<br>Equivalent values allowed for input.</td><td rowspan="2">Task conformance<br>The degree to which the system services supports all the tasks the user wishes to perform and the way in which the user understands them.</td></tr>
<tr><td>Consistency<br>Likeness of behaviour arising in like situations.</td><td>Customisability<br>Modifiability of the user interface by the user or system.</td></tr>
</table>

**Table 5.5 – Usability principles according to Dix *et al.* [19].**

## 5.6 Developments in Reporting Exceptions

As motivated in Section 5.2 the reporting of exceptions influences the usability of the system and is exceedingly important. Exception reporting mechanisms are difficult to implement and application dependent. Provision must be made for different types of users and their different needs. Also, due to the construction of systems, the feedback provided is either static or pre-determined and not dynamically conversational in a current situation.

A more recent approach to address the issue of continuous and dynamic feedback to different users is the *Hercule* system [64]. *Hercule* consists of an exception-reporting manager that is able to dynamically interpret machine application states and to translate the states to visual communication for the various users [66]. It is a generic application, thus application independent, which observes the interaction between the application and the environment and provides feedback based on this information as active dialogue to the user.

Exception reporting is, like most other sub-disciplines in software development, steadily becoming a specialised area and *Hercule* is the first step towards creating a generic facility [65] that can be reused with different applications and that addresses the problems concerning continuous, dynamic and conversational feedback. Another advantage of *Hercule* is that it does not require, like static reporting, code duplication [65]. It can also be used independently of the development process.

*Hercule* consists of an independent window that is divided into sections for the display of different types of information. Figure 5.6 gives an example of the *Hercule* window [64]. At the top left hand side it indicates the system's readiness with a traffic light anomaly, the green circle at the bottom indicating "ready" and the red circle at the bottom indicating, "finished". At the bottom left it provides functionality for providing context with the function "Replay My Actions". On the right hand side feedback is provided on what is currently happening [64].

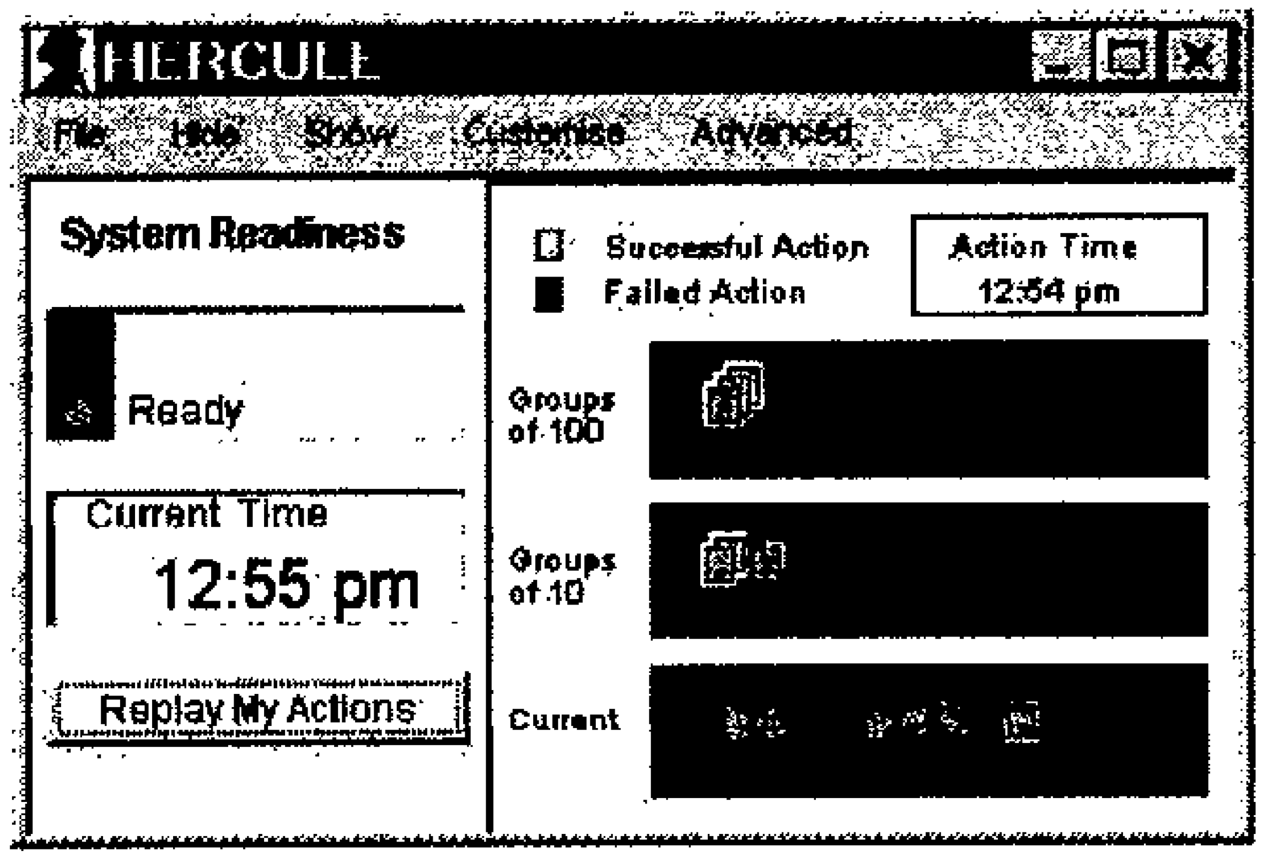



**Figure 5.6 – The *Hercule* window [64].**

Another feature of *Hercule* where it improves on other schemes it the ability to provide different types of feedback for users in different roles. Examples of the different types of feedback can be seen in Figures 5.7 – 5.9 [64]. Figure 5.7 depicts the feedback for the end-user giving a description of the system action in understandable language, Figure 5.8 feedback for the programmer or developer in terms of details concerning internal functioning and Figure 5.9 feedback for systems support teams in terms of performance.

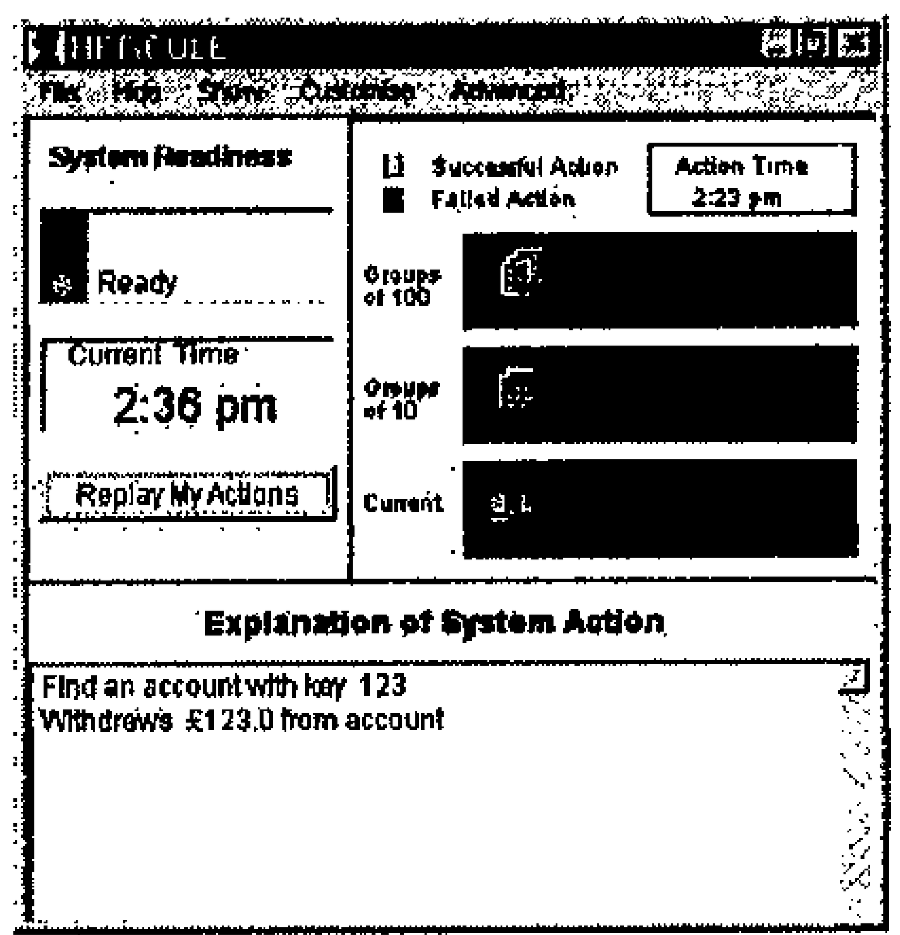



**Figure 5.7 – User feedback**

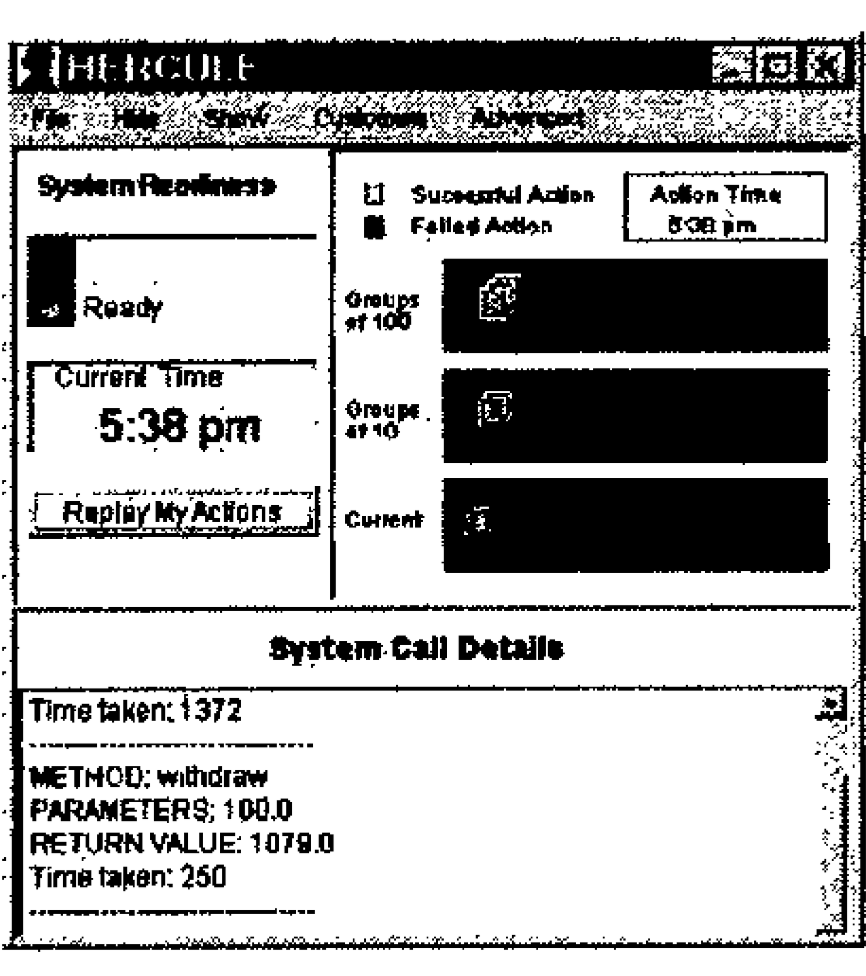



**Figure 5.8 – Developer feedback**

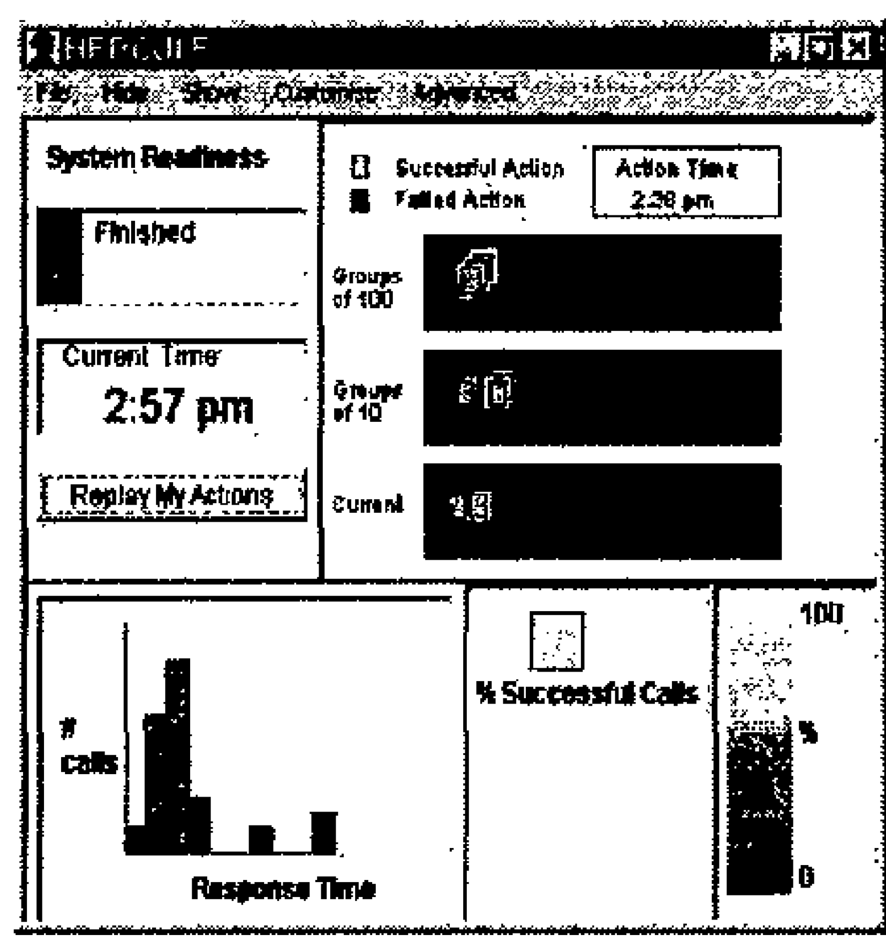



**Figure 5.9 – Support feedback**

*Hercule* might be able to address the problems surrounding exception reporting as it

- includes the information necessary for adequate exception reporting,
- provides different forms of feedback for the different types of users,
- takes account of different levels of understanding of the different users,
- reports continuously and immediately as requested,
- provides details and context,

- makes use of a graphical, uniform display for reporting throughout the application, and
- distinguishes between normal and exceptional behaviour.

## 5.7 Summary

Section 3.1 indicated that the successful handling of exceptions, rather than the occurrence of exceptions, should be used as an indication of the quality of a software system. The understanding that is achieved in this chapter re-enforces what was stated in Section 3.1. The major conclusion being that the degree of usability of a system in the case of an exception is proportional to the degree of successful communication. The system communicates or reports exceptions to the user by means of messages. The success of the communication is reliant on the adequacy of these messages. As systems advance and become more sophisticated the user will expect the same sophistication of the exception reporting mechanisms. The provision of dynamic feedback and active dialogue will become more important as they serve to provide better communication [65]. Independent feedback generators such as *Hercule* might prove to be the answer to successful reporting of exceptions.

Section 5.2 described the necessity of the reporting of exceptions, Section 5.3 the type of information that needs to be included when reporting on exceptions and Section 5.4 described how they should be reported. Section 5.5 mentioned the principles from Dix *et al.* [19] to support usability and Section 5.6 introduced *Hercule*, a new approach to exception reporting.

# Chapter 6 - Proposals

## 6.1 Introduction

The purpose of this chapter is to formulate a view on the current status of exception handling in all its facets as considered in the previous chapters. Section 6.2 summarises the different aspects of the problem areas encountered and investigated. Section 6.3 lists and describes proposed solutions from the literature to address some of the problems and Section 6.4 follows with the requirements extracted from these proposed solutions for a solution to address most of, if not all, the problem areas. Section 6.5 rounds off with a short summary.

## 6.2 Identified Problem Areas

This section gives a summary of the contents of Chapters 2 – 5 in terms of the different aspects of exceptions. A brief description is given of what has been reported and the remaining concerns.

In Chapter 1 it was noted that exceptions are a reality that must be dealt with as they can have serious repercussions on the user, the system and the environment and that the reliability and quality of software is influenced by the occurrence of exceptions. There is a need for more reliable software - thus better handling of exceptions and current methods seem inadequate judging by the number of failures still experienced.

Chapter 2 investigated the causes and origins of exceptions, software testing and testing techniques. The conclusion drawn is that although different testing techniques are applied during the different phases of the software development life cycle to eliminate the occurrence of exceptions in operational systems, they cannot eliminate all exceptions or guarantee that all exceptions will be handled successfully.

The various aspects of the exception life cycle were investigated in Chapter 3 - from detection to handling. The major concern arising from each of these aspects is that a continuous flow of information is needed between the phases. Information about the decisions taken and the rationale for the decisions should be transferred from one phase to the next to ensure the

consistent and uniform addressing of exceptions and the implementation of exception-handling mechanisms throughout the software system.

Language implementation details were investigated in Chapter 4 and although the languages provide enough exception-handling features to be used for the development of operational software there are limitations. Firstly, all the features must be implemented as prescribed and as intended in order to be able to function correctly and secondly they provide only limited functionality for some of the aspects. Flexibility is lacking in the sense that the languages are developed with certain models as foundation and do not always provide support for other models.

While Chapter 4 reported on the internal details of exception handling Chapter 5 investigated the reporting of an exception occurrence externally by means of messages. The conclusion drawn from this chapter is that the reporting of exceptions is necessary and that effective communication requires a mechanism capable of reporting exceptions dynamically.

In addition to the problems mentioned above Robillard and Murphy [70] also found that the exception-handling structure degrades over time, as does the normal behavioural structure due to maintenance. The main reasons they found:

- There are unpredictable sources of exceptions, by which it is implied that the exceptions are undetected until they occur. It is impossible to test every possible implementation and interaction scenario and therefore untested sources of exceptions will always remain.
- Unanticipated exceptions, exceptions expected not to occur, that occur.
- Handler overload – due to their hierarchical-type structure, and in order to reuse them, exception handlers handle too many exceptions that are only vaguely related.
- Propagation of exceptions leads to the loss of context as they are forwarded any number of times to any number of handlers and with every propagation the relationship between the exception and the handler weakens.
- Exception overload. Quite different exceptions might be represented by a single exception due to inheritance.
- Systematically ignoring exceptions and postponing the actual addressing of the exception.

- Unspecified exception values such as "Exception 1". The meaning of these values is lost and the effort needed to rectify the situation increases as time passes. As a result the situation is often never rectified.
- Inconsistent use of exception-handling mechanisms.

The next section will look at proposals from the literature to address some of these problems.

## 6.3 Existing Proposals

This section describes a few of the proposed solutions to some of the problems addressed in Section 6.2. These models do not address identical problems and as such cannot be compared, but they all have merit and contain ideas that might lead eventually to a sought-after, uniform, exception-handling framework. This framework could then be implemented in an attempt to solve the problems and limitations currently experienced. All these models do, however, have two similarities. Firstly, they all attempt to define an exact but distinct exception-handling mechanism, separate from the rest of the system, so as to isolate but fully integrate the mechanism in modular form. Secondly, they all strive to handle exceptions in a conceptually more uniform, structured and manageable way, irrespective of the type of exception or application.

Section 6.3.1 deals with the proposals surrounding the representation of exceptions and Section 6.3.2 deals with proposals that apply to the dynamic management of exception occurrences in a system.

### 6.3.1 Representation of Exceptions as Objects [15, 26, 20, 51]

The idea is to view and deal with an exception as an object. Exceptions can then be incorporated within the development process like other objects and the same established notation and concepts used.

Moessenboeck *et al.* [51] proposed that exceptions should be defined as objects of a class called the exception-class. The exception-class is a subclass of the class *exception*. Exceptions are categorised as either system or user exceptions. Diagram 6.1 illustrates the concept. Dony [20] proposes the same type of idea with his proposal of meta-classes. The occurrence of exceptions is instances of a subclass of the meta-class *ExceptionalEvent* that owns the common behaviour and

predetermines the basic structure of exceptions. *ExceptionalEvent* is further divided into *FatalEvent* and *ProceedableEvent* to distinguish between those that can be handled and those that can not. Garcia *et al.* [26] explore the same idea when they propose the representation of exceptions as data objects in a hierarchical structure with *Exception* being the root of the structure and exception handlers as methods.

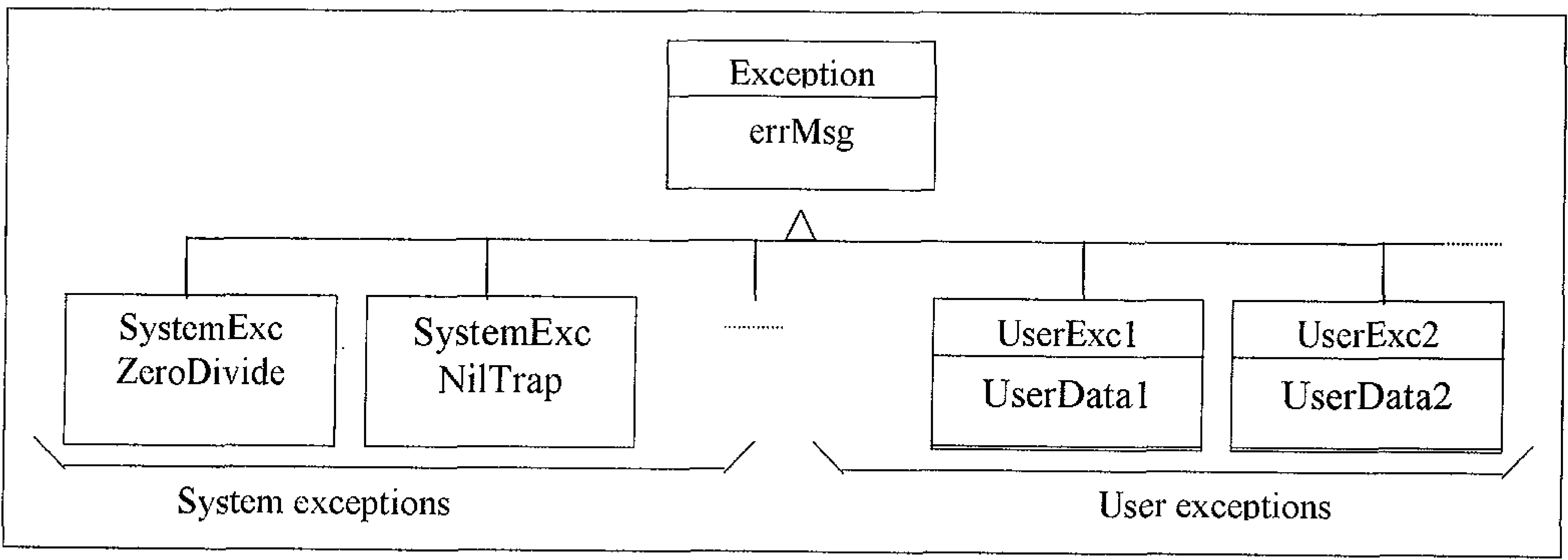



**Diagram 6.1 – Exceptions as objects [51].**

The advantages of the representation of exceptions-as-objects are that:

- Additional overhead is only created if and when an exception occurs. In the course of normal processing the exception-handling code is not executed. This supports the concept of orthogonality mentioned in Section 4.2 [26, 51].
- It is conceptually easy to implement [26, 51] and provides a meaningful structure for dealing with exceptions.
- It provides for the uniform handling of system-defined as well as user-defined exceptions, in both cases exceptions are signalled and handled by the system in the same manner [20, 26]. Unexpected exceptions can also be handled by defining a "universal exception" in the same manner as the rest of the exceptions.
- It serves to separate normal from exceptional execution [26] thus modularity is encouraged.
- It is easily extensible, with modifications such as adding, removing or changing a type of exception done easily with the aid of the hierarchy [20, 26].
- Reuse is possible with the aid of the hierarchical structure [20, 26]. It allows for a single handler to catch any exception that is related to a set of exceptions, meaning that even unexpected exceptions could be handled by a single handler.

- Propagation is done according to the hierarchy and invocation chain [20, 26]. A class controls which exceptions will be propagated to a higher level up the hierarchy. A method of a class (implementation of a handler) can handle all exceptions propagated to it by lower levels and provide context dependent information [20].
- A handler can be attached at different levels [20, 26], expressions, methods, objects and classes making it flexible. Handlers can be designed for - and associated with - different granularities - on a high level with classes and on a lower level with expressions. This ensures that the implementation can match the conceptual modelling of responsibility assignment more closely. Responsibilities can vary with regard to the object or the class responsible for an action, as discussed in Section 3.4.3.2
- All the handling models, as mentioned in Section 3.5, namely, terminate [26], resume and retry can be implemented by methods [20]. A choice can also be provided between these models to add flexibility [20].
- This idea does not require any specific language constructs, which makes it language independent and it thus lends itself to easy implementation in an object-oriented language [20, 26, 51].
- It can be implemented in both centralised, distributed and concurrent systems [26] since communication can be established via message passing.

The disadvantage of the representation of exceptions-as-objects is that it requires more design and coding effort. However, cost-benefit analysis indicates that the effort and costs are worthwhile if compared to the alternative and its problems.

To summarise - the representation of exceptions-as-objects lends itself to the incorporation of exception handling throughout the software development life cycle, hand-in-hand with the development of the non-exceptional behaviour of the system. It supports the implementation of the React-Repair Policy in order to create reliable software and it deals practically with the issues involved with this policy as mentioned throughout Chapter 3.

### 6.3.2 Adding an Exception Manager

A very different approach to exceptions is the addition of an exception manager whose responsibility it is to oversee interaction between the user and the system, report exceptions and manage the exception-handling processing.

Various proposals have been put forward regarding systems that acquire information about them selves and use the information to change processing dynamically [15, 26]. This is often called reflection and is implemented in order to progress from restrictive exception handling and its limitations, to an approach that is more flexible and matches more closely the realistic experience with environments.

Most systems are divided into layers, levels or tiers to support object-oriented development and communication in the client – server model [65]. The upper layer consisting of the interface forms the client and a lower level, the server, provides the functions requested by the client. The "manager" is positioned between the interface layer and the processing layer and oversees the processing of the system. Each layer can consist of many components. Diagram 6.2 depicts the concept diagrammatically in a simplified form.

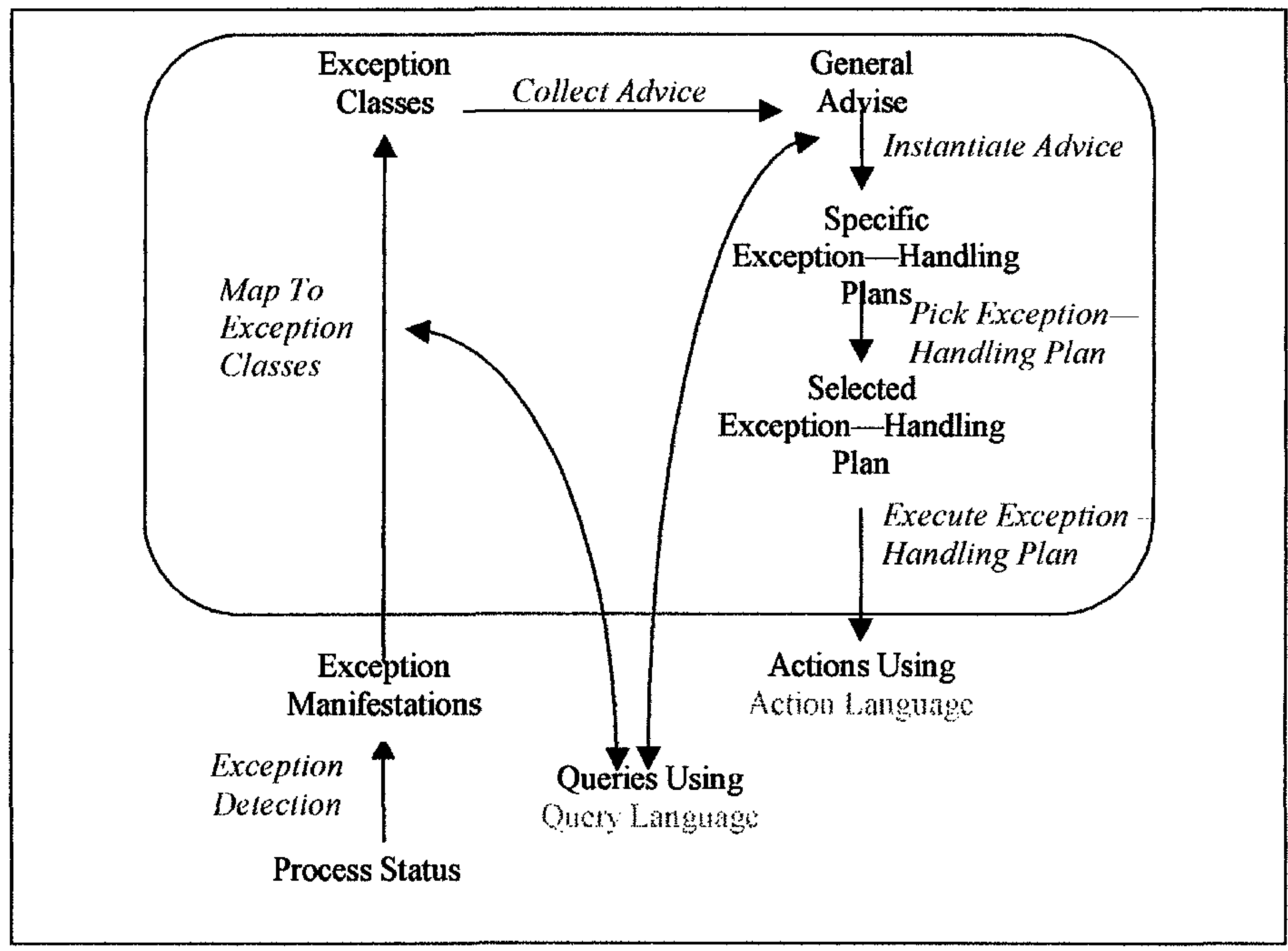



**Diagram 6.2 – The functionality of the "exception-manager" [37].**

A few authors have proposed related ideas: Robillard and Murphy [70] with what they call compartments, Dellarocas [18] with component-based systems and Klein and Dellarocas [37]

with agent-based systems. The idea is to develop a system as a number of interacting components by concatenating the foregoing ideas. Interaction between components would be restricted to make the complexity of interaction and recovery more manageable in the event of an exception. The words *compartment* and *component* will be used interchangeably.

There is no real restriction on what a component can be, but it serves the purpose of specifying boundaries for the propagation of exceptions [70]. Only propagation of specific exception classes is allowed – and this serves to improve robustness [70]. The component is formed with an interface and a description of the exceptions that may propagate from it [70], with the component itself considered to be implementing only "normal" behaviour [18, 37]. By means of the interface it can report on its state and its behaviour is also amended via the interface [18, 37].

A separate exception-handling service, permanently implemented (the "manager"), uses these interfaces to detect exceptional behaviour. Detection is done by analysing behavioural patterns [18]. Exceptional behaviour is then typed by traversing the taxonomy of possible types while presenting the effects of the exception as well as information about the process model [37]. On typing the exception the possible resolutions can be identified and a decision made as to the appropriate one for implementation.

This exception-handling service is shared by all the components [37] and makes use of a knowledge base and exception-management expertise to invoke appropriate corrective actions. It has knowledge about exceptions and their manifestations in the system and can provide, by means of the knowledge base, specific resolutions derived from generalised resolutions [37].

*Hercule,* as described in Section 5.6, is an example of a prototype of such an exception-manager that aims to provide meaningful feedback to the user [66]. The developers of *Hercule* found that component-based systems introduce problems of their own. These problems are due to the different and independent origins of the components [65]. Different components tend to have different conventions regarding the handling of exceptions, as they are developed by different developers who do not always interact to establish standard conventions for the issues surrounding the handling of exceptions [65]. What makes *Hercule* different from Dellarocas's [18] system is that it is *independent* of the application and acts as an intermediary between the system and the user, not as an agent between other agents. It observes the interaction between the

application and the user to provide the user with feedback [65]. Figures 6.3 and 6.4 depict the position of *Hercule* in relation to the rest of the system.

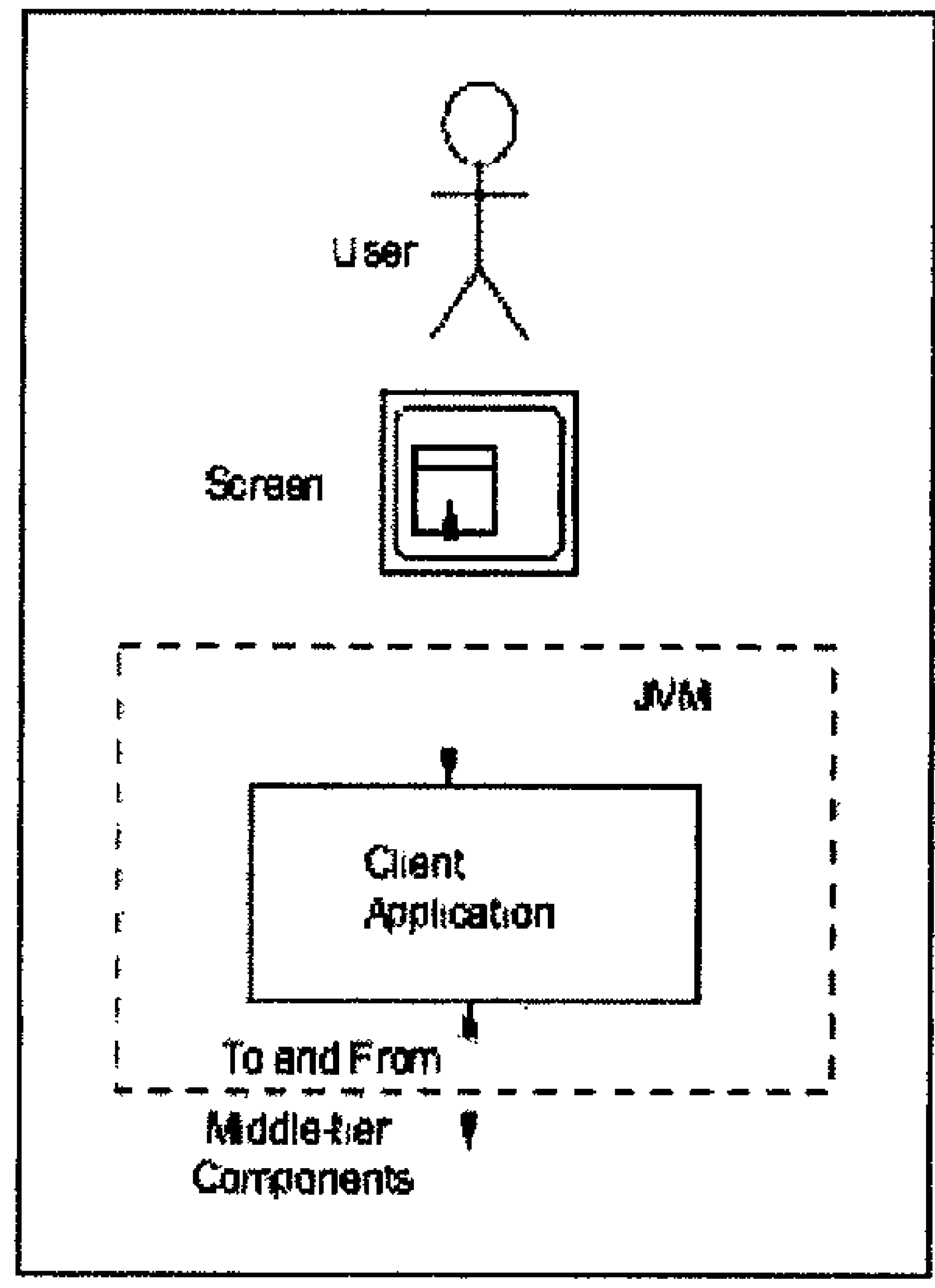



**Figure 6.3 – Layered software.**

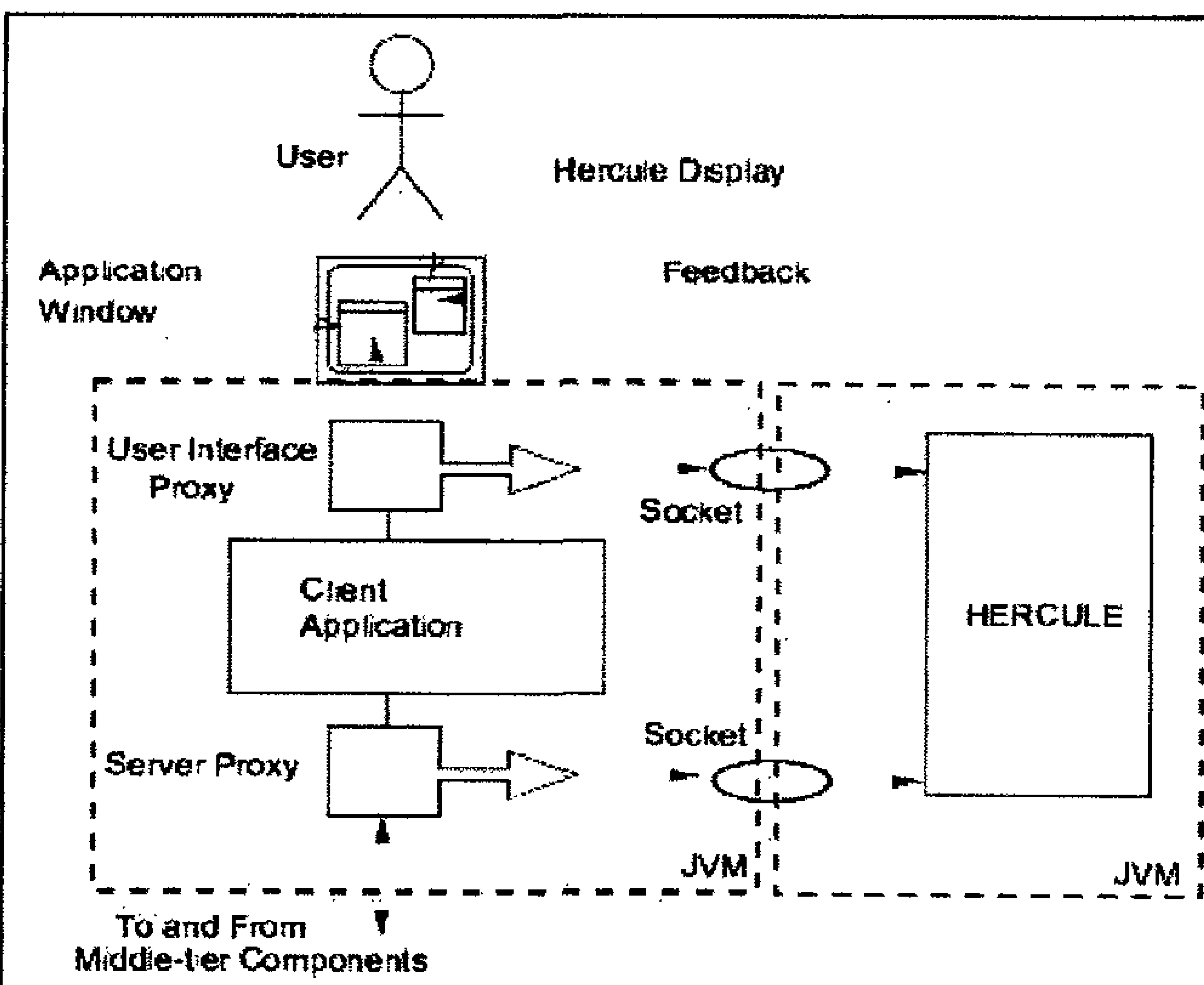



**Figure 6.4 – *Hercule* positioned.**

Advantages of the added exception manager:

- Separates the handling of exceptions from normal processing [37]. It also serves the purpose of making complexity more manageable [18].
- Allows for reuse since it makes use of generic exception handling and heuristic classification where expertise is separated from knowledge applied [37].
- More effective and easier to modify due to the modularity of the design. It is ideal for an environment that is complex, dynamic and error-prone. It can also be extended since it is possible to add or "plug in" services into a system.
- Exception handlers are searched dynamically using the type of exception as criterion [51]. It allows dynamic modification [15], which makes it very flexible and assists the modelling of changing and fluctuating systems. It allows for automated or human-computer co-operation in the exception-handling process [15, 37, 46, 67]. The difference is only in the expertise required. As mentioned by Borgida [8] humans are better qualified to adapt to exceptional conditions and make decisions about them than are application programs.

Disadvantages of the added exception-manager:

- The overall complexity increases due to the fact that exception handling is often not easily localised in a particular component and that each component might be involved in many different exceptions [37].
- It is difficult to build a context for exception reporting.

The addition of an exception-manager lends itself to dynamic exception handling and models more closely the human capabilities of exception handling, which is superior to that of current static software implementations. The enhanced flexibility allows for a closer modelling of real-world situations.

Existing problems have been listed, as have the solutions that have been proposed to address some of the problem areas. However, not all problem areas are covered and the proposed solutions address only specific issues. The next section will investigate the form of a solution that will address most, if not all, the problem areas. It lists the requirements of such a solution, the roles it has to fulfil in the software development life cycle as well as the improvements and advantages of such as solution and the remaining limitations and disadvantages. The final subsection will discuss the development and role of patterns in relation to the solution and exception handling in general.

## 6.4 Proposal for a Unified Exception-Handling Framework

It is clear from the problems listed in Section 6.2 and the solutions proposed in Section 6.3 that a perfect solution to the exception problem has not, as yet, emerged. The identified problem areas indicate that there is a definite need for a solution and the proposals indicate a definite effort by various researchers to search for such a solution.

In order to improve the exception-handling capabilities of a system, *all* aspects concerning exceptions must be addressed. It is of little use if only one aspect improves. For example, if the detection capabilities of a system are improved but the system has poor handling mechanisms the worth of the improved detection capabilities is nullified. A unified exception-handling framework must encapsulate all the aspects of the exception life cycle in relation to the overall software development life cycle. Such a framework will aid exception handling throughout the software

development process in order to improve overall exception handling and minimise the effects of exceptions. It will assist developers in making structured decisions about various aspects of exceptions and provide practical and proven guidelines for structured thinking about exceptions.

The following quotations emphasise the need for a unified exception-handling framework:

- *"Related work in this area has been scarce, but most of this work associates exception handling only with objects and makes no attempt in considering a unified framework for representing exception-handling in the software life cycle"* [17].
- *"No systematic methodology is available to help developers identify all the possible exception types and appropriate resolution strategies"* [18].
- *"One of the distinguishing characteristics of intelligent human behavior in the natural world is flexibility, the ability to deal with unusual, atypical, or unexpected occurrences. In this regard, however, practically all current commercially developed software is the exact antithesis of the human system that it replaces or assists: programs are extremely rigid and intolerant of deviations from the norm set out by their designer"* [8].
- *"Different strategies for exception handling can be found in today's object-oriented languages (OOL) but no standard solution exists"* [20].
- *"Even though many object-oriented languages provide exception –handling facilities, a few of them provide an exception mechanism that is really integrated with the object model"* [26].
- *"Ideally it should be coherent with the language, the entire programming paradigm, and the design methodology"* [85].

Having established the need for a unified exception-handling framework the next step is to identify the requirements of such a framework. This is done in Section 6.4.1. These requirements can then be used to evaluate proposed frameworks.

### 6.4.1 General Requirements

The following requirements emerge from the literature: (References in italics indicate that the idea originated from the literature but that the reference is a modification or extension of the original idea and should not be taken as a literal reference.)

- They must be practically and consistently [58] applicable throughout the whole of the software development life cycle [9], be language independent, and applicable to all types of systems – centralised, concurrent and distributed. They must have a structured

framework enforcing controlled and disciplined application [9, 46, 58, 85]. They must also make use of a minimum notation to convey maximum information [58].

- They must be able to detect, identify, handle and report all types of exceptions uniformly [9] but at the same time prevent the continuation of incomplete or inconsistent processes and components. Provision should also be made for the handling of unexpected exceptions.
- They must allow for the separation of exception-related processing from normal processing facilitating understanding and design and should allow for the explicit change in flow of control to support this separation. Exception propagation should be provided for to facilitate the allocation of a handler in the situation where the handler first assigned cannot handle the exception.
- They must allow for modularity in design to minimise coupling in order to allow for reuse, extensions and modifications [13] increasing the ease of maintenance.
- They should be flexible [9, 58] by making provision for deviations and various possibilities where it exits, without overloading. Mechanisms should be put in place that allow for evolution in systems in correspondence to the evolution of the underlying organisation [16]. More emphasis should be placed on the co-operative resolution of exceptions between humans and systems [76].
- They should facilitate the effective and dynamic reporting of exceptions to different types of users.
- They should alleviate testing throughout the system by improving readability and programmability [13].

Having completed this list of requirements the following section will consider how exception handling can be incorporated throughout the software development life cycle.

### 6.4.2 Inclusion of Exception Handling in the (Iterative) Software Development Life cycle

Borgida [8] is of the opinion that exception handling should be deferred and that there should be an initial focus on establishing the normal processing requirements of the software. He states that this adheres to the principle of abstraction whereby less important aspects of the systems are at first ignored in order to make complexity manageable. Special, rare or unusual cases should be deferred. He proposes that even during the design and implementation process exception-

handling should be deferred and added in the form of annotations later, even more so in relation to the development of databases. The advantages would be easier development, maintainability and improved readability due to the fact that the exception-handling issues are kept separate from the rest of the system development.

This point of view is definitely not popular due to the problems experienced with it as mentioned in the literature and quoted earlier in this dissertation. Exceptions can definitely not be viewed as less important aspects of a system due, to the severity of the effects they might have on such a system. It has become clear during the course of this research that different aspects of exception handling need to be attended to during the different phases of the software development life cycle [17]. To keep up with continuously evolving requirements, exceptions' requirements need to evolve as well. Exceptions should not be viewed in isolation from the rest of the system, since they are an inherent part of the system. Khriss *et al.* [34] amplify this with their statement that *"Software development raises the need for traceability, i.e. the ability to control the consistency between software artefacts produced at different stages of the software life cycle."* The earlier fault tolerance is addressed the greater the potential protection against the specification faults, but the higher the cost in terms of the redundancy of software [53].

The following list sets out what should be done in the earlier development phases in order to incorporate exception handling and to establish traceability throughout the life cycle as a consequence:

- *Specification and Requirements Analysis Phase:*
    1. A policy statement should be included to set the goal of exception handling in the system [4].
    2. The policy should define in broad terms, the acceptable behaviour of the system *[4]* at an abstract level *[46]* to complement the character of this development phase.
    3. The policy will form the basis of an exception handling mapping between phases *[46]*.
    4. It should specify what the system should, and should not do in terms of normal and exceptional behaviour [52].
- *Analysis Phase:*
    1. The policy statement must be extended by making use of strategies that will make it realistically implementable. The strategies specify what must be done when an

exception occurs, i.e. what is the proposed course of action from the detection to the handling of the exception [52].

2. Exception sources should be identified in terms of faults, errors, failures and constraints *[46].* They can then be described and classified in terms of abstract properties [17].

- *Design Phase:*
  1. During the design phase the details for detection and reaction must be set out [52].
  2. An exception hierarchy can be set up in order to associate handlers with the types of exceptions.
  3. Exceptions should now be described in terms of messages and interfaces [17] and assigned responsibilities [4].

Diagram 6.5 illustrates the incorporation of exception handling in the earlier phases of the development life cycle [4]:

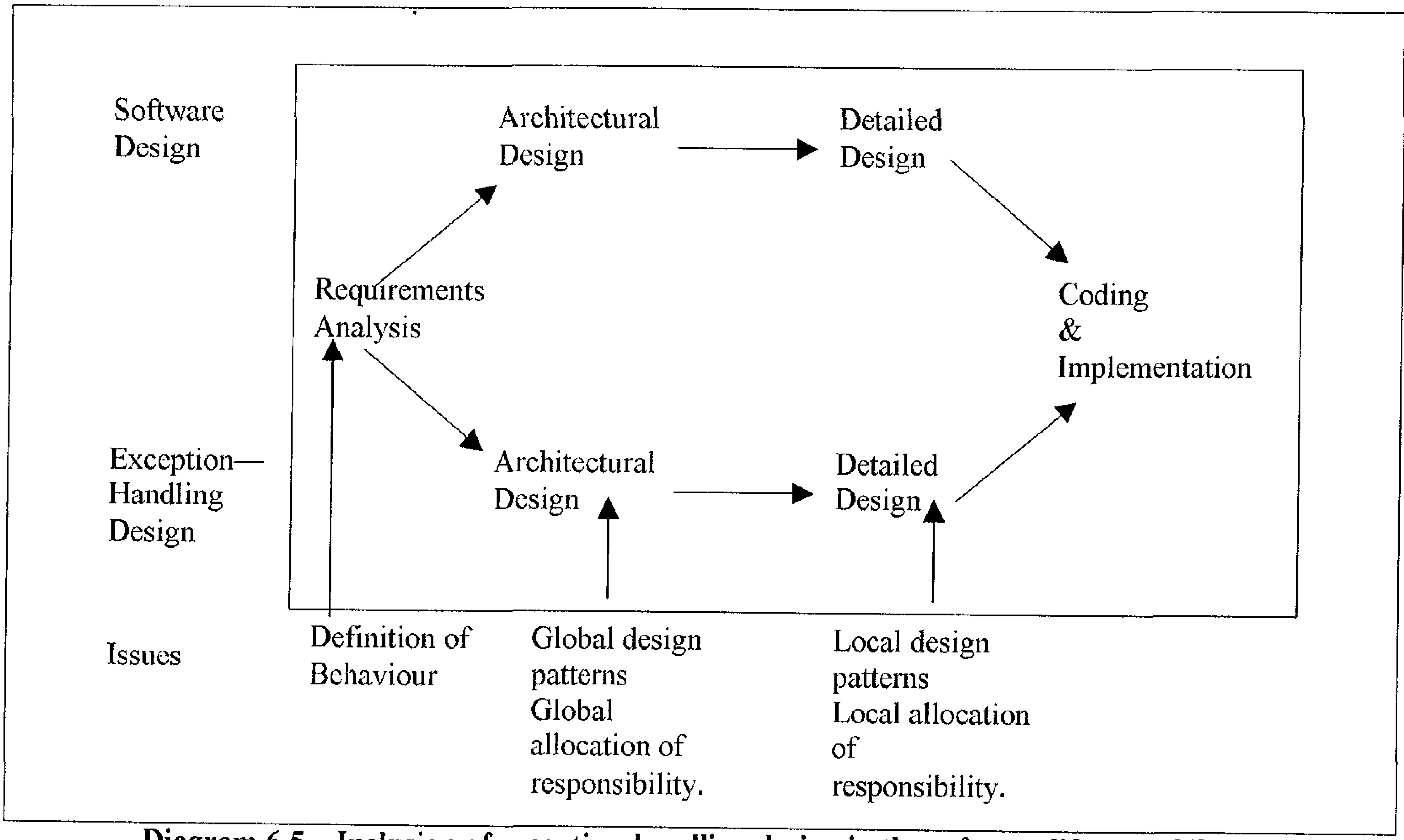



**Diagram 6.5 – Inclusion of exception-handling design in the software life cycle [4].**

### 6.4.3 Evaluating the Unified Exception-Handling Framework

Any emerging framework must be evaluated, since the quality of the software developed with the aid of the framework will be influenced by the quality of the framework itself. Sections 6.4.3.1

and 6.4.3.2 evaluate the proposal for a framework in terms of its possible improvements and limitations.

#### 6.4.3.1 Improvements and Advantages

The improvements and advantages listed here indicate in which areas the framework will benefit the quality of software development and, as a consequence, the quality of the end product.

- It will provide traceability of exception handling throughout the whole of the development life cycle.
- It will increase the ease of maintenance, if for instance, a developer or programmer leaves the system, the expertise and the work done will be documented for those responsible for maintenance.
- Testing will be better facilitated and assurance costs reduced.
- Quality will be improved through increased availability.

The second most important advantage, besides traceablity, is that experience will be gained and through this experience refinements can be made as insight is gained. Experienced developers will generate patterns and inexperienced developers will benefit from them. The available knowledge regarding exceptions will become more structured and will increase.

#### 6.4.3.2 Limitations and Disadvantages

- Increased complexity will result.
- Implementation overhead will be increased.
- Languages may not be able to support the framework's models and the models will have to be adapted.
- The occurrence of unexpected exceptions cannot be eliminated.
  There is always the possibility that an unexpected exception will arise. The framework cannot guarantee successful handling of *all possible* unexpected exceptions. The only way to deal with them is to create a universal handler for "unexpected exception".

#### 6.4.3.3 Summary

The advantages of the framework outweigh the disadvantages. First of all, the goal of exception handling is to develop more reliable software and the framework addresses the problems currently experienced. More reliable software will result if the problems that have been identified are

addressed. Secondly, since the complexity and the implementation overhead that is introduced is less than that introduced by modifying the software after it becomes operational, it is worthwhile to invest in the additional overhead. Lastly, limitations introduced by languages can be overcome and as for unexpected exceptions, a universal handler can be provided, which, although not a perfect solution, it is better than having no handler at all.

### 6.4.4 The Role of Patterns in the Unified Exception- Handling Framework[1, 4, 25]

Patterns identify, name and abstract common themes in object-oriented development. Exceptions are an example of such a theme and patterns can prove to be useful in dealing with exceptions. A pattern is usually applied during the design phase, and can be described as a rule that consists of three parts, *"which expressed a relation between a certain context, a certain system of forces which occurs repeatedly in that context, and a certain software configuration which allows these forces to resolve themselves"*[1]. Patterns aim to capture and compile knowledge and experience about recurring problems [1].

The general requirements discussed in Sections 6.4.1 would lead to the eventual creation of exception-handling patterns that would serve as proven "rules-of-thumb". This source of knowledge and experience would also aid novice developers, who would learn from the experience of the experts [25]. Besides capturing knowledge and experience, patterns also provide the following additional advantages [25]:

- Patterns facilitate communication as they can be documented and referenced by name.
- A reusable library of knowledge and experience is formed for building sound and reusable software.
- They reduce the learning time needed for novices. Novices don't have to gain experience by trial and error. It takes time to gain experience, but the transformation process from novice to expert status is faster if patterns are utilised.

Although it is necessary to know what to do, it is just as important to know what to avoid. Patterns are used to indicate positive rules, i.e. "do's", and anti-patterns to indicate negative rules, i.e. "don't s". It is very easy to identify when a pattern or solution is *not* working, which gives rise to an anti-pattern. Anti-patterns can be seen as extensions of patterns that focus instead on a *"selection of repeated software failures in an attempt to understand, prevent and recover from*

*them*" [12] Anti-patterns start with an existing negative solution, i.e. a solution that is *not* working, in order to recognise and better understand the situation, symptoms and consequences of the underlying problem area [12].

Ineffective exception-handling design patterns are listed by [4]. There are two main categories of these exception-handling patterns, namely exception-based and feature-interaction-based:

- *Exception based* – those that misuse a language's exception mechanisms.
    - *Predefined exceptions:*

      These should be used properly to preserve their effectiveness and consistency. They should only be raised implicitly and never be renamed.
    - *Exception Handlers:*

      Change in the flow of control in a handler should be done explicitly and when a handler does not propagate an exception the values of out or in-out parameters must set or reset to prevent the calling subprogram from attempting to use erroneous values. Null handlers, handlers with no specified actions, are not desired. Inadvertent mappings – a user defined exception raised but potentially overridden by a local exception-handling response, nested block statements and default handlers should also be approached with care: nested block statements, because of the complexity involved, and default handlers due to the lack of exception processing.
    - *Propagation*:

      Propagating exceptions can create "anonymous exceptions"; those propagated out of scope that leads to a loss of context.

- *Feature-interaction-based* – those that interact with other language mechanisms to corrupt the exception-handling processing.
    - *Generics*:

      Exception declaration should be done properly, making use of unique or common names depending on the granularity of the name space. Elaboration features – dynamic generic instantiations, result in more elaboration failures than non-generic units.

- *Derived Types*:

  Exceptions propagated by implicitly declared subprograms are the same as exceptions propagated by explicitly declared subprograms.
- *Tasking*:

  In the case of multi-path propagation, responsibility for actions must be allocated appropriately. Task termination must be made known if an exception propagates from it, to ensure a consistent view of the task's state by the system. A task should not exit before dependent tasks are terminated.
- *Optimisations*:

  Optimisation features of compilers might affect the exception-handling behaviour.
- *System level features*:

  Additional care should be taken if return codes are used. The suppression of range checks can lead to problems.
- *Recursion*:

  Handlers performing indirect recursive calls should be avoided.

## 6.5 Summary

The unified exception-handling framework was introduced in this chapter as a proposal for a solution to encapsulate and address the problems currently experienced surrounding exceptions and the handling of exceptions. If such a framework can be developed and utilised it will definitely, despite the very real disadvantages, ensure the development of more reliable software. The main feature of the framework would be the provision of traceability of an exception's life cycle throughout the software development life cycle. Secondly, it would provide a structure for the thinking and decision-making surrounding exception handling, which would ensure that exceptions are not dealt with in an *ad hoc* fashion.

This chapter concluded the research reporting, with Section 6.2 summarising the current problem areas concerning exceptions. Section 6.3 described some of the current solutions proposed to address some of the problem areas and Section 6.4 investigated the requirements of a near-perfect solution that addresses the problems – the unified exception-handling framework. In the next chapter, Chapter 7, a brief overview of the major contents of the dissertation is provided.

# CHAPTER 7 – CONCLUSION

## 7.1 Introduction

This dissertation researched exceptions in relation to the software development life cycle, as shown in Diagram 7.1.

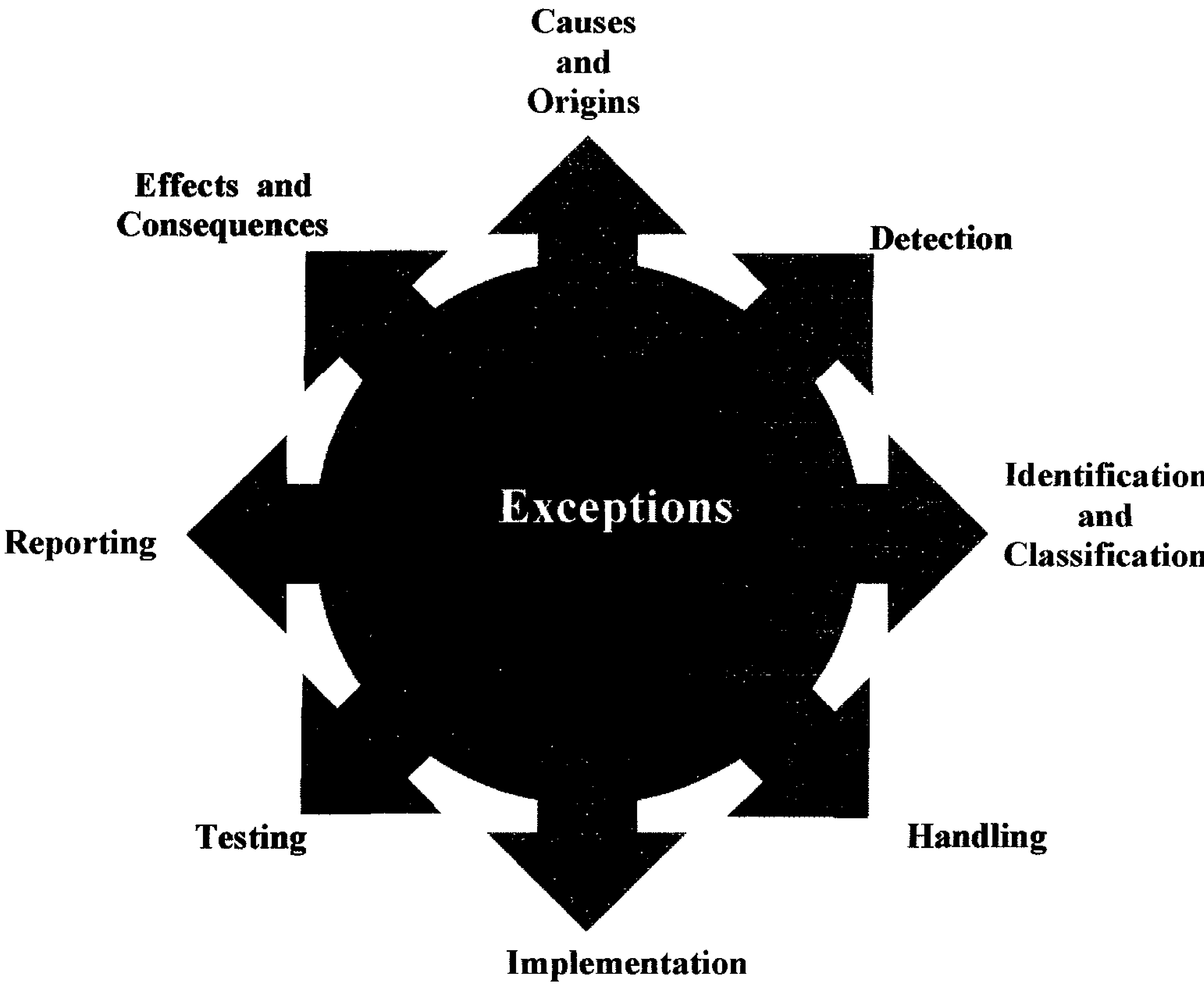



**Diagram 7.1 – A graphical depiction of the issues concerning exceptions as discussed in the dissertation.**

The problem areas surrounding exceptions were summarised and proposals made to address some of the problem areas. The conclusion drawn was that the issues are real, that there is a definite need to solve or at least address some of the issues, and that research is currently being done in this area.

## 7.2 Review

Chapters 2 to 5 confirmed what was stated in Chapter 1 - that in the first place, exceptions are a reality that must be dealt with and that in the second place, there are some serious issues surrounding exceptions. These issues cannot be ignored as they can seriously affect systems and those who interact with them.

Chapter 2 discussed the testing techniques used to investigate the origins and causes of exceptions and to determine why it is not possible to create exception-free software. Testing techniques are only an aid to eliminating exceptions where possible, and cannot be held responsible for the exceptions that remain in software products. Testing techniques are successful in uncovering exceptions and defective exception-handling mechanisms but cannot guarantee the elimination of *all* exceptions. Testing techniques will improve and new techniques will be developed to uncover even more exceptions but the total elimination of exceptions will never be possible. The effects of exceptions remaining and occurring in operational software on the user, the system and the environment were also investigated in Chapter 2.

Chapter 3 described the well-researched aspects concerning the life cycle of an exception from detection through to handling. It also motivated the need for exception handling to be incorporated throughout the software development life cycle to establish trace-ability, or a "story-line", through all the phases of the development life cycle and concluded that the structures to do so were not yet in place. It could also be established that there are already a number of well-defined policies and that there are strategies to implement them. It became clear that structures and notations were needed to wed the implementation of stated policies and the incorporation of such policies from the early development phases onwards. In order to do so, a classification scheme for exceptions is needed, which is also not yet in place.

Chapter 4 made it clear that object-oriented languages make provision for the implementation of the aforementioned handling models. However, the successful implementation of exception-

handling functionality as provided by the language itself, relies not only on the correct implementation in code but also the correct interpretation of its use by the programmer. There is undoubtedly room for improvement, but the development of languages should not be done independently of the modelling paradigm and without taking the analysis and design issues mentioned into account. Languages developed independently of the modelling paradigm - and issues relating to analysis and design - will never be able to address these issues successfully.

Reporting the occurrence of an exception for the user of the system is of great importance. Chapter 5 considered this issue. The user must, at all times, be aware of the state of the system and even more so when an exception has occurred. Even if it is not possible to resolve an exceptional situation, it must be reported in such a manner that the user has a clear idea of what has happened, why it has happened, how to prevent such exceptions in future, and what to do next.

Chapter 6 made a proposal to solve some of the problems identified in the previous chapters. The main conclusion being that dealing with exceptions should not be confined to the last stages of the software development life cycle and that it should be incorporated within the whole of the development process as echoed here by the following quotations:

- *"Consequently, in the cost-effective engineering of reliable software, it can be appropriate to supplement fault prevention with design approaches which attempt to suppress the effects of residual faults"* [53].
- *"Thus, the higher the level at which fault tolerance is applied, the greater the change of faults that are addressed and, in particular, the greater the potential protection against specification faults, but the higher the cost in terms of software redundancy"* [53].

This chapter presented the most significant contribution of this dissertation - namely a proposed structure for the incorporation of exception-handling policies throughout the software development life cycle.

## 7.3 Future Research

The direction for future research might be to develop mechanisms to incorporate and make provision for exceptions throughout the whole of the software development life cycle. The framework should be general enough to be able to be incorporated with the existing analysis and design languages, for example, UML, with a notation complementing the existing notations but

different enough to be distinct. The main criteria being that the framework must be practical to implement, without adding complexity to the already complex development process and should be functional. Such a unified exception- handling framework should integrate with the system and environment, and mimic the exception-handling capabilities of humans.

# Appendix A – Exception- Handling Issues In Concurrent Distributed Systems.

## A.1 Introduction

Many organisations require applications that operate concurrently from different locations due to the nature and physical distribution of the organisation [13]. The issues that were mentioned in Chapters 2 - 6 are applicable to centralised as well as distributed systems. The complexities of distributed, and possibly concurrent systems, necessitate additional structures to make these complexities manageable [13]. Distribution refers to the physical distribution of resources, and concurrency to multiple components possibly executing simultaneously to reach a common goal [84]. These structures need additions to the normal object-oriented paradigm to extend this paradigm and make it more applicable for the development of concurrent and distributed systems. It also extends the functionality of existing object-oriented languages whereby these conceptual constructs can be implemented in the languages.

The increased complexity in these systems is due to the possibility of:

- independent exceptions being raised concurrently [84],
- exceptions raised asynchronously while interacting with different components,
- multiple related exceptions being raised concurrently,
- delay or lack of communication,
- exceptions occurring in different parts or locations of the system [84, 85].
- real-time systems adding additional time constraints [13]
- exceptions generated due to the distributed nature of the system, e.g. network failures.

If provision is not made to detect and handle concurrent exceptions in a coordinated fashion some of the exceptions might pass through the system unhandled [13, 84, 85].

It is impossible to manage the complexity of any number of components interacting simultaneously in any possible combination with one another. One possibility put forth by Xu *et al.* [85] is, in order to be able to manage the complexities involved, to restrict communication between components with the use of coordinated atomic actions (CA Actions) [84]. This has a restricting effect on the development and design [71] of the system but the increase in ability to

manage the complexities cancels this disadvantage. The restriction of interaction improves the ability to recover from exceptions [85].

Sections A.2 and A.3 describe two approaches for managing the complexities of exception handling in concurrent and distributed systems.

## A.2 Coordinated Atomic Actions (CA Actions) [35, 85]

This concept is a scheme for coordinating complex concurrent activities, and supports recovery from exceptions between multiple interacting objects in a distributed object-oriented system [71]. It provides for the handling of cooperative and competitive concurrency [35].

A CA Action co-ordinates recoveries from the occurrence of exceptions between multiple interacting and interdependent components (or objects) by integrating the ideas of conversations and transactions [85]. Concurrent processes can either be independent, competing or co-operating, implying that the actual threads executing can respectively either access a set of objects disjointedly, not disjointedly or collectively.

An atomic transaction is an operation seen as a single entity that is committed, if executed successfully, or rejected, if not executed successfully. It ensures atomicity, consistency, isolation and durability (the "ACID" properties) [35, 71, 85]. In the case of an unsuccessful operation, the effects of the transaction are reversed, as if the transaction were never attempted, with backward error recovery. To summarise, a transaction ensures consistency of shared resources in the presence of failures due to exceptions and competitive concurrency [35, 71].

A conversation defines, and thus restricts, the processes that are allowed to communicate and exchange information [84]. Each conversation has a boundary and no information exchange is allowed to cross the boundary [71]. When a process enters a conversation its state is saved and it only communicates with components or processes participating in the same conversation. An acceptance test is used to establish if the transactions were successful, if not all the processes were rolled back to the saved state, thus backward error recovery. Asynchronous entry into the conversation is allowed but exiting is done synchronously at the time of the passing of the acceptance test [85]. To summarise, a conversation controls co-operative concurrency and implements coordinated error recovery [35, 71].

Through the use of these concepts the context, declaration and propagation of exceptions in concurrent and distributed systems can be clarified [85]. Distinct nodes in different locations have to communicate via message passing since they have separate memory and disjoint address spaces [75, 84]. The time required for messages to be sent and received and delays in this regard may not be ignored [84, 85]. Delays may allow for corrupt information to be spread throughout the system causing numerous related exceptions in a wide variety of locations. Diagram A.1 illustrates the concept of a CA Action [85].

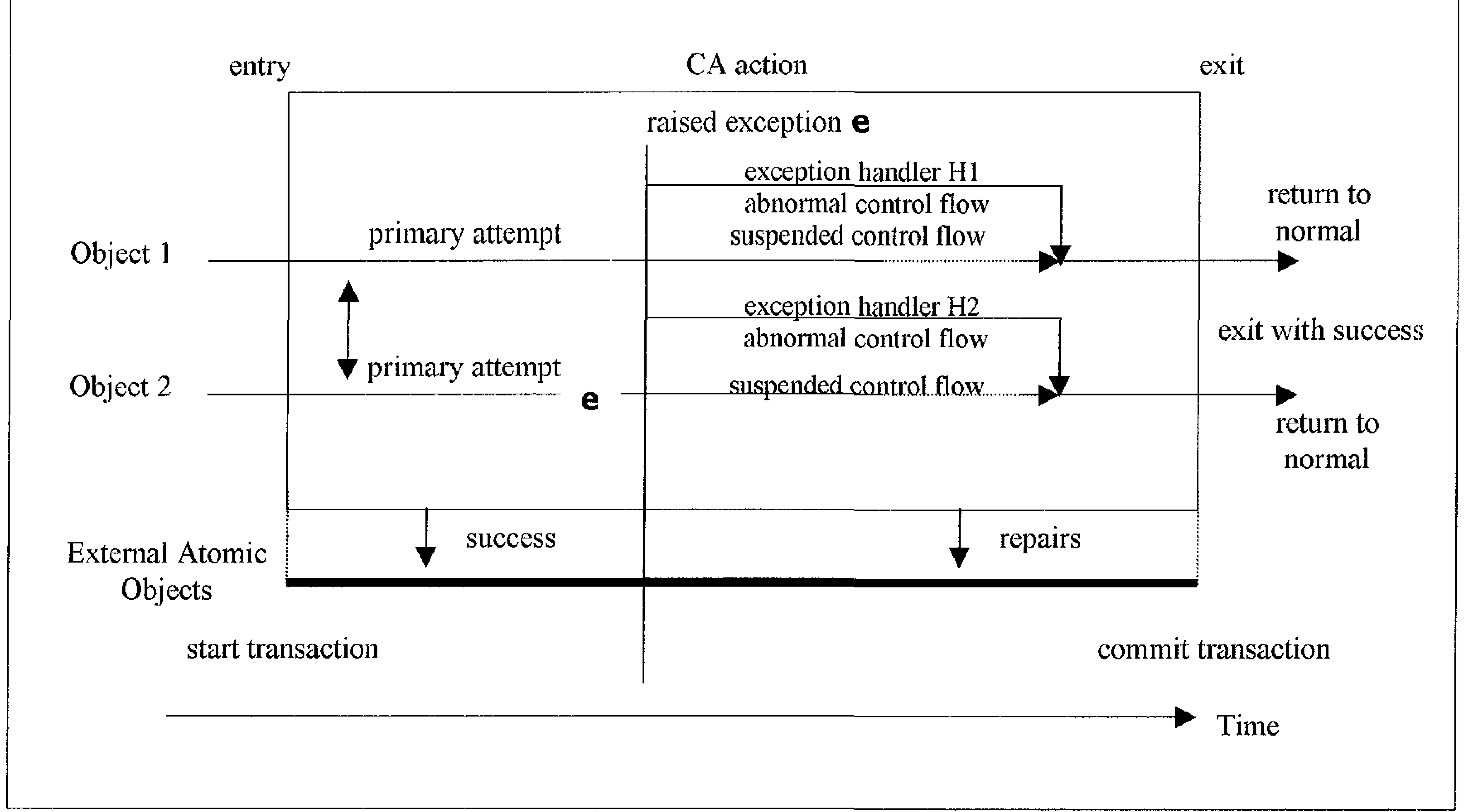



**Diagram A.1 - The concept of the CA Action [85].**

A set of exceptions is associated with each conversation and each process in the conversation owns a set of handlers for some or all of these exceptions [84]. If an exception is raised, the appropriate handlers for the same exception in all the processes are executed and the processes co- operate to recover [84, 85].

In order to curb the occurrence of multiple related exceptions occurring, an exception tree is used. This is a hierarchical tree structure depicting and ordering all the exceptions associated with a conversation and their relationships. An exception handler can handle an exception at a level and

those at lower levels, but not at higher levels [84, 85]. The universal exception is at the root of the tree.

### A.2.1 Blocking versus Non- Blocking Communication [13, 71]

Communication must take place in order to inform the participants in a conversation of the occurrence of an exception. There are two options blocking and non-blocking (pre-emptive) communication.

In the case of blocking communication, the sender blocks until the receiver is ready to receive a message [13]. It either reaches the end of an action or encounters an exception. In the case of an exception, the information is sent to all the participants and the sender can only receive messages from other participants and their states after it has completed sending [71]. Blocking communication is easier to implement since each process is ready for recovery and in a consistent state when a handler is called, but it causes delays and might lead to deadlocks [71].

In the case of non-blocking communication the sender does not block [13]. The action detecting the exception interrupts the other participants in the case of an exception [71]. In this instance no time is wasted but it is far more difficult to analyse, understand and improve and the abortion of nested programs is very difficult to program [71].

In both cases the receiver is blocked if the sender has yet to send an event [13].

### A.2.2 The Necessity of Concurrent Exception Resolution

For the following reasons that form the core of the differences between a centralised and concurrent- distributed system, the additional features are necessary [85]:

- It is difficult to interrupt all the operations of all the participating objects immediately after the occurrence of an exception. A measure is needed to prevent the continuation of execution of processes with erroneous values leading to new exceptions being raised due to the lack of information about the current exception.
- The period of time elapsing between messages being sent and received also cannot be ignored. It can have the same effect as mentioned above, erroneous values spreading and leading to more exceptions being raised [84].

- Due to the nesting of CA Actions and the time interval between message-passing, concurrent exceptions can be raised. Queuing might be an option here, queuing the events and servicing them in a specified order [13].
- Different objects may be involved at different levels in the conversation and their context would be different as to the exception occurring.
- The probability of hardware related exceptions is higher. One option might be to convert hardware interrupts to language events and treat them as asynchronous events [13].
- The probability of misunderstandings and inconsistencies in the development and design is higher.

## A.3 Co- operative Object- Oriented Approach [74]

This is a variation of CA actions, designed especially to address the issues relating to the development of complex concurrent systems throughout the whole of the software development life cycle. The handlers are specified, and as the classes extend with the addition of new exceptions so the handlers are related to more exceptions. During each iteration more exceptions are identified. During each phase and iteration failure assumptions are also made and revised as the development evolves.

It represents the co-operation between objects at different phases of the software life cycle. The aim is to incorporate the description of exceptional behaviour in both objects and co-operations, from the early phases. Since it is a structured approach, the intention is that it should alleviate the complexity of the design.

Objects alone are insufficient to describe system behaviour, thus this approach defines systems by their components and the relationships between components

A template for describing an object class for the early phases of the life- cycle [74] can be found in Figure A.2:

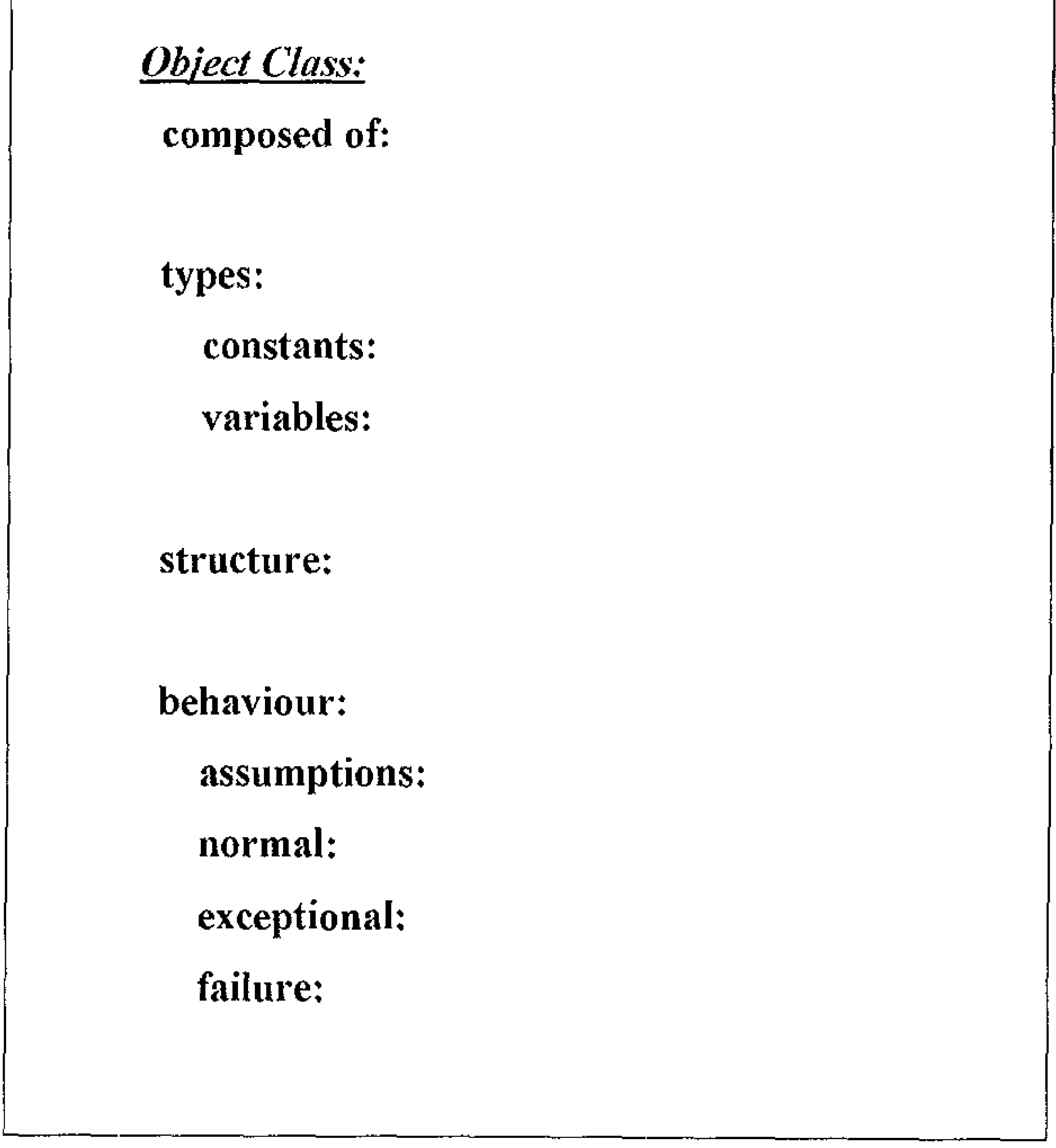



**Figure A.2 - Template for describing an object class [74].**

The role of a CO action is to co-ordinate collaboration between objects. This includes coordinating the handling of exceptions between co-operating objects. There are two scenarios that must be taken into account, either an object can handle the exception locally, or not. In the latter scenario it must be propagated to co-operating objects in a coordinated manner.

In the early life cycle, objects and their co-operation are represented in terms of their properties, whilst later in the life- cycle in terms of messages and interfaces. In the design phase a CO action is depicted as an object class. A CO action can either be implemented as a separate object class or distributed between objects, which participate in co-operation.

A template for describing a CO action in the early phases of the life- cycle [74] can be found in Figure A.3:

*CO Action Class:*

**participants:**

**types:**

**constants:**

**variables:**

**behaviour:**

**normal:**

**exceptional:**

**failure:**

**Figure A.3 - Template for describing a CO action [74].**

The behaviour specification of both objects and CO action are defined in terms of the event-action model which provides a set of concepts for the modelling and analysis of phenomena associated with the system and its environment.

The specification of exceptional behaviour for objects and CO actions requires the specification of an exception as an event, in terms of system variables representing an error in the system state. The specification of the exception handler is an action in terms of the action's respective start and finishing events. The system predicates describe the start event of a handler are associated with the error recovery activities which put the object of CO action back into a known consistent state. This is captured by the system predicates describing the finished event of the handler. In terms of the CO action the system predicate of the finish event is associated with the post-condition of the exception handling, which represents possible degraded behaviour of the action.

*"The main intent behind the cooperative object- oriented approach for software development is to use modified versions of structuring mechanisms as modelling abstractions that can be employed throughout the software lifecycle"* [74].

## A.4 Summary

Sections A.2 and A.3 briefly discussed additional concepts that were introduced in order to make the inherent complexity of concurrent and distributed systems more manageable. There are other concepts being developed based on those mentioned, which will attempt to manage the complexity with less restrictions on the interactions, such as Open Multithreaded Transactions [35].

# References

## Articles & Books:


[1] B. Appleton. 2000.
*Patterns and Software: Essential Concepts and Terminology.*
**[W1] Last Accessed on 15/10/2001.**

[2] M. H. Austern. 1998.
*Making the World Safe for Exceptions*
C++ Report. Vol. 10. No.1. pp. 30-39.

[3] R. L. Baber. 1991.
*Error-Free Software. Know-how and Know-why of Program Correctness.*
John Wiley & Sons. pp. 1-11.

[4] W.G. Bail. 1999.
*Exception Handling Design Patterns.*
Advances in Computers. Vol. 49. pp. 191-238.

[5] E. Bertino, S. De Capitani Di Vimercati, E. Ferrari and P. Samarati. 1998.
*Exception-Bases Information Flow Control in Object-Oriented Systems.*
ACM Transactions on Information and System Security. Vol. 1. No.1. pp. 26-65.

[6] S. Biffl and W. Grossman. 2001.
*Evaluating the Accuracy of Defect Estimation Models Based on Inspection Data from Two Inspection Cycles.*
23rd International Conference on Software Engineering pp. 145-154.

[7] E. Börger and W. Schulte. 2000.
*A Practical Method for Specification and Analysis of Exception Handling – A Java/JVM Case Study.*
IEEE Transactions on Software Engineering. Vol. 26. No. 9. pp. 872-887.

[8] A. Borgida 1985.
*Language Features for Flexible Handling of Exceptions in Information Systems.*
ACM Transactions on Database Systems. Vol. 10. No.4. pp. 565-603.

[9] A Borgida and T Murata. 1999.
*Tolerating Exceptions in Workflows: A Unified Framework for Data and Processes.*
Software Engineering Notes. Vol. 24. No. 2. pp. 59-68.

[10] C. Bramford and B. Dollery. 1996.
*OODREX: An Object-Oriented Design Tool for Reuse with Exceptions.*
OOIS '95 International Conference. pp. 248-251.

[11] P. J. Brown. 1983.
*Error Messages: The Neglected Area of the Man/Machine Interface?*
Communications of the ACM. Vol. 26. No.4. pp. 246-249.

[12] W. Brown, R. Malveau, H. McCormick III, T. Mowbray and S.W.Thomas. 1998.
*Anti-Patterns.*
John Wiley and Sons Inc.
**[W7] Last Accessed on 16/10/2001.**

[13] P. A. Buhr and W.Y. R.Mok. 2000.
*Advanced Exception Handling Mechanisms.*
IEEE Transactions on Software Engineering. Vol. 26. No.9. pp. 820-831.

[14] T. Cargill. 1996.
*Exception-Handling: A False Sense of Security.*
C++ Gems. Edited by Stan Lippman.
Cambridge University Press.

[15] D.K.W. Chiu, Q. Li and K. Karlapalem. 1999.
*A Meta Modeling Approach To Workflow Management Systems Supporting Exception Handling.*
Information Systems. Vol. 24. No.2. pp. 159-184.

[16] CSST Technologies, Inc. 1997.
*A Defect Classification Framework.*
TEST-Rx™ Standardized Software Testing Process Methodology.
**[W5] Last Accessed on 20/05/2001.**

[17] R. de Lemos and A. Romanovsky. 1999.
*Exception Handling in a Co-operative Object-Oriented Approach.*
Proceeding of the 2nd IEEE International Symposium on Object-Oriented Real-Time Distributed Computing. pp. 3-13.

[18] C. Dellarocas. 1998.
*Toward Exception Handling Infrastructures for Component-Based Software.*
1998 International Workshop on Component-Bases Software Engineering. pp. 1-8.

[19] A. Dix, J. Finlay, G. Abowd and R. Beale. 1998.
*Human Computer Interaction*
*Second Edition*
Prentice-Hall. pp. 162-175.

[20] C. Dony. 1990.
*Exception Handling in Object-Oriented Programming: Towards a Synthesis.*
ECOOP/OOPSLA '90 Proceedings. pp. 322-330.

[21] R. T. Douglas. 1990.
*Error Message Management. Automating error message documentation.*
Dr. Dobb's Journal. Jan 1990. pp. 48-51.

[22] T. J. Duesing and J.R. Diamant. 1997.
*CodeAdvisor: Rule-Based C++ Defect Detection Using a Static Database.*
Hewlett-Packard Journal. Vol. 48. No. 1. pp. 19-21.

[23] M. S. Feather. 1996.
*Modularized Exception Handling.*
SIGSOFT '96 Workshop. pp. 167-171.

[24] B. Fields, P. Wright and M. Harrison. 1996.
*A Method for User Interface Development in Safety-Critical Applications.*
UNISA 16-20 September 1996.

[25] E. Gamma, R. Helm, R Johnson and J. Vliissides. 1993.
*Design Patterns: Abstraction and Reuse of Object-Oriented Design.*
ECOOP '93.

[26] A.F. Garcia, D. M. Beder. and C.M.F. Rubira 1999.
*An Exception Handling Mechanism for Developing Dependable Object-Oriented Software Based on Meta-Level Approach.*
10th IEEE International Symposium on Software Reliability Engineering.

[27] J. B. Goodenough. 1975.
*Exception Handling: Issues and a Proposed Notation.*
Communications of the ACM. Vol. 18. No.12. pp. 683--695.

[28] T. Guinther. 1998.
*Between Java and C++ ...*
EXE. Vol. 12. No.8. pp. 57-58.

[29] C. Hagen and G. Alonso. 2000.
*Exception Handling in Workflow Management Systems.*
IEEE Transactions on Software Engineering. Vol. 26. No.10. pp. 943-958.

[30] W. Harrison, C. Barton and M. Raghavachari. 2000.
*Mapping UML Designs to Java.*
ACM Sigplan Notices. Vol. 35. No.10. pp. 178-187.

[31] R. Helm. 1995.
*Patterns in Practice.*
Proceedings of the 10th Annual Conference on Object-Oriented Programming Systems Languages and Applications.

[32] R. Hower.. 1996-2001.
*Software QA/Test Resource Center.*
*Software QA and Testing Frequently-Asked-Questions Part 1.*
**[W2] Last Accessed on 16/10/2001.**

[33] R. Hower.. 1996-2001.
*Software QA/Test Resource Center.*
*Software QA and Testing Frequently-Asked-Questions Part 2.*
**[W2] Last Accessed on 16/10/2001.**

[34] I. Khriss, R.K. Keller and I.A. Hamid 2000.
*Pattern-based Refinement Schemas for Design Knowledge Transfer.*
Knowledge-Based Systems. Vol. 13. No.6. pp. 403-415.

[35] J. Kienzle. 2000.
*Exception-Handling in Open Multithreaded Systems.*
Exception Handling in Object-Oriented Systems Workshop at ECOOP'2000.

[36] M. Klein. 1995.
*Conflict Management as Part of an Integrated Exception Handling Approach.*
Artificial Intelligence for Engineering Design, Analysis and Manufacturing.
Cambridge University Press, USA. Vol. 9. pp. 259-267.

[37] M. Klein and C. Dellarocas. 1999.
*Exception Handling in Agent Systems.*
Proceedings of the 3rd Annual Conference on Autonomous Agents. pp. 62-68.

[38] J. L. Knudsen. 2000.
*Exception-Handling versus Fault Tolerance.*
Exception Handling in Object-Oriented Systems Workshop at ECOOP'2000.

[39] A. Koenig. 1997.
*Partial Library Implementations and Type Conversions.*
C++ Report. Vol. 9. No. 10. pp. 12-13.

[40] P. Koopman and J. DeVale. 2000.
*The Exception Handling Effectiveness of POSIX Operating Systems.*
IEEE Transactions on Software Engineering. Vol. 26. No.9. pp. 837-848.

[41] D. R. Kuhn. 1999.
*Fault Classes and Error Detection Capability of Specification–Based Testing.*
ACM Transactions on Software Engineering and Methodology. Vol. 8.No.4. pp. 411-424.

[42] L. Lamport. 1979.
*On the Proof of Correctness of a Calender Program.*
Communications of the ACM. Vol. 22. No.10. pp. 554-556.

[43] C. Larman. 1998.
*Applying UML and Patterns.*
Prentice Hall.

[44] C. Lewis, P. Polson, C. Wharton and J. Rieman. 1990.
*Testing a Walkthrough Methodology for Theory-Based Design of Walk-Up-and-Use Interfaces.*
Conference on Human Factors and Computing Systems. pp. 235-242.

[45] S. B. Lippman. 1991.
*C++ Primer. 2nd Edition.*
*Appendix B: Exception Handling.*
Addison-Wesley Publishing Company. pp. 571 – 578.

[46] Z. Luo, A. Sheth, K.Kochut and J. Miller. 2000.
*Exception Handling in Workflow Systems.*
Applied Intelligence. Vol. 13. No.2. pp. 125-147.

[47] D. Mandrioli and B. Meyer. 1991.
*Advances in Object-Oriented Software Engineering*
*Chapter 1: Design by Contract.*
Prentice Hall. pp 18-38.

[48] M.M. Manning. 1997.
*Teach Yourself Borland J Builder in 21 Days.*
*Week 2, Day 13.*
Sams.net Publishing. pp. 459 – 473.

[49] R.A. Maxion and R.T. Olszewski. 2000.
*Eliminating Exception Handling Errors with Dependability Cases: A Comparative, Empirical Study.*
IEEE Transactions on Software Engineering. Vol. 26. No.9. pp. 888-906.

[50] B. Milewski. 1997.
*Resource Management in C++.*
Journal of Object-Oriented Programming. Vol. 10. No.1. pp. 14-22.

[51] H. Moessenboeck, M. Hof and P. Pirkelbauer. 1997.
*Zero-Overhead Exception Handling Using Metaprogramming.*
Proceedings SOFSEM '97. Nov. 1997. Lecture Notes in Computer Science.

[52] B. Morgan *et al.* 1997.
*Microsofts Visual J++.*
*Chapter 7: Advanced Java Programming.*
Sams.net Publishing. pp. 144-162.

[53] M. Moulding. 1990.
*Software Reliability Handbook.*
*Chapter 4: Software Fault Tolerance.*
Edited by P. Rook.
The Universities Press(Belfast) Ltd. pp. 83-110.

[54] T. Murata and A. Borgida. 2000.
*Handling of Irregularities in Human Centered Systems: A Unified Framework for Data and Processes.*
IEEE Transactions on Software Engineering. Vol. 26. No.10. pp. 959-977.

[55] J. Nielsen. 1995.
*Usability Inspection Methods.*
CHI '95 Mosaic of Creativity. pp. 377-378.

[56] J. Neilsen. 2001.
*Error Message Guidelines. Summary.*
pp. 1-2.
**[W6] Last Accessed on 10/10/2001.**

[57] D. Orchard. 1998.
*Better Performance with Exceptions in Java.*
BYTE (International Edition). Vol. 23. No.3. pp. 53-54.

[58] K. T. Owens. 1998.
*Building Intelligent Databases with Oracle PL/SQL, Triggers and Stored Procedures Chapter 10: Error Handling and Exceptions.*
Prentice-Hall. pp 299-332.

[59] R. F. Paige, J.S. Ostroff and P. J. Brooke. 2000.
*Principles for Modeling Language Design.*
Information and Software Technology. Vol. 42. No. 10. pp. 665-675.

[60] S. Park, J. Kim and S. Lee. 2000.
*Agent-Oriented Software Modeling with UML Approach.*
IEEE Transactions on Information Systems. Vol. E38-D. No. 8. Pp. 1631-1641.

[61] D.E. Perry, A. Romanovsky and A. Tripathi. 2000.
*Current Trends in Exception Handling.*
IEEE Transactions on Software Engineering. Vol. 26. No.9. pp. 817-819.

[62] R. S. Pressman.. 1992.
*Software Engineering. A Practitioner's Approach.*
*Third Edition.*
McGraw-Hill Inc.

[63] J. W. Reeves. 1997.
*Reflections on Exceptions.*
C++ Report. Vol. 9. No.3. pp. 57-65.

[64] Karen Renaud
*Hercule* (Link)
**[W8] Last Accessed on 16/10/2001.**

[65] K. Renaud. 2000.
*A Generic Feedback Mechanism for Component-Based Systems.*
University of Glasgow. Ph. D. Thesis. pp. 104-111.

[66] K. Renaud and R. Cooper.
*Augmenting Feedback by Visualizing Application Activity.*

[67] K. Renaud and R. Cooper. 2001.
*Considering Possible Outcomes and the User's Environment in Designing User Interfaces for Data-Intensive Systems.*
UIDIS 2001.

[68] K. Renaud, P Kotzé and T. van Dyk. 2001.
*A Mechanism for Evaluating Feedback of E-Commerce Sites.*
I3E-Conference.

[69] R.Rist and R. Terwilliger. 1995.
*Object-Oriented Programming in Eiffel.*
*Chapter 17: Exceptions.*
Prentice-Hall, Australia. pp. 329 – 343.

[70] M. P. Robillard and G. C. Murphy. 1999.
*Regaining Control of Exception Handling.*
**[W4] Last Accessed on 19/10/2001.**

[71] A. Romanovsky, B. Randell, R. Stroud, J. Xu and A. Zorzo. 1997.
*Implementation of Blocking Coordinated Atomic Actions Bases on Forward Error Recovery.*
Journal of Systems Architecture. Vol. 43. No.10. pp. 687-699.

[72] J.L. Schilling. 1998.
*Optimizing Away C++ Exception Handling*
Sigplan Notices. Vol. 33. No.8. pp. 40-47.

[73] D. C. Schmidt. 1996.
*Using Design Patterns to Guide the Development of Reuseable Object-Oriented Software.*
ACM Computing Surveys. Vol. 28. No. 4es. pp 162-166.

[74] P. J. Schroeder and B. Korel. 2000.
*Black-Box Testing Reduction Using Input-Output Analysis.*
ISSTA 2000. pp. 173-177.

[75] R. Stewart, S Bhattacharya and D. Rai. 1998.
*Distributed Exception Handling in COBRA-Bases C++ Applications.*
C++ Report. Vol. 10. No.3. pp. 32-43.

[76] D. M. Strong and S. M. Miller. 1995.
*Exceptions and Exception Handling in Computerized Information Processes.*
ACM Transactions on Information Systems. Vol. 13. No.2. pp. 206-233.

[77] H. P. Sutter. 1997.
*More Exception-Safe Generic Containers.*
C++ Report. Vol. 9. No.10. pp. 24-32.

[78] R.K. Swadley et al. 1996.
*Delphi 2 Unleashed.*
*Chapter 6: Exceptions.*
Sams Publishing. pp. 165 – 193.

[79] P. Thomas, H. Robinson and J. Emms. 1988.
*Abstract Data Types, Their Specification, Representation and Use.*
*Chapter 4: Exception Handling.*
Oxford University Press. pp 109-115.

[80] M. Tsagias and B. Kitchenham. 2000.
*An Evaluation of the Business Object Approach to Software Development.*
Journal of Systems and Software. Vol. 52. No.2-3. pp. 149-156.

[81] A. von Mayrhauser, R. France, M. Scheetz and E. Dahlman. 2000.
*Generating Test-Cases from an Object-Oriented Model with an Artificial-Intelligence Planning System.*
IEEE Transaction on Reliability. Vol. 49. No.1. pp. 26-36.

[82] J. Waldo. 1997
*Taking Exception*
UNIX Review. Vol. 15. No.7. pp. 69-73.

[83] C. Weir. 1998.
*Code that Tests Itself... Using Assertions and Invariants to Debug Your Code.*
C++ Report. Vol. 10. No.4. pp. 26-31.

[84] J. Xu, B. Randell, and A. Romanovsky. 1998.
*Coordinated Exception Handling in Distributed Object Systems: From Model to System Implementation.*
Proceedings of the 18th International Conference. (Cat. No. 98CB36183) pp. 12-21.

[85] J. Xu, A. Romanovsky and B. Randell. 2000.
*Concurrent Exception Handling and Resolution in Distributed Object Systems.*
IEEE Transactions on Parallel and Distributed Systems. Vol. 11. No.10. pp. 1018-1032.

[86] S. Yemini and D. M. Berry. 1985.
*A Modular Verifiable Exception Handling Mechanism.*
ACM Transactions on Programming Languages and Systems. Vol. 7. No.2. pp. 214-243.

## Electronic Media – The Internet

[W1] http://www.enteract.com/~bradapp/docs/

[W2] http://www.softwareqatest.com

[W3] http://www.acm.org

[W4] http://citeseer.nj.nec.com

[W5] http://www.csst-technologies.com/TEST-Rx.html

[W6] http://www.useit.com/alertbox

[W7] http://www.antipatterns.com

[W8] http://www.dcs.gla.ac.uk/~karen/hercule.html