



Thesis submitted to the

UNIVERSITY OF COIMBRA

for the partial fulfillment of the requirements for the degree of
Doctor of Philosophy in Informatics Engineering

A Transactional Model For Automatic Exception Handling

Bruno Miguel Brás Cabral

Under supervision of

Prof. Paulo Jorge Pimenta Marques
Dep. de Engenharia Informática
Universidade de Coimbra, Portugal

**Departamento de Engenharia Informática
Faculdade de Ciências e Tecnologia
Universidade de Coimbra**

Julho 2009



**Departamento de Engenharia Informática
Faculdade de Ciências e Tecnologia
Universidade de Coimbra**

**ISBN 978-989-96001-1-9
Coimbra - Portugal, Julho 2009**

This investigation was partially supported by the Portuguese Research Agency - FCT, through a scholarship (SFRH/BD/12549/2003), by CISUC (R&D Unit 326/97), by POCI 2010, and by the European Social Fund

 **Ciência.Inovação 2010** Programa Operacional Ciência e Inovação 2010
MINISTÉRIO DA CIÊNCIA, TECNOLOGIA E ENSINO SUPERIOR



— *To my lovely wife*—

Abstract

Exception handling mechanisms have been around for more than 30 years. Although modern exceptions systems are not very different from the early models, the large majority of modern programming languages rely on exception handling constructs for dealing with errors and abnormal situations. Exceptions have several advantages over other error handling mechanisms, such as the *return of error codes* or the usage of *global state flags*. Exceptions eliminate, for instance, the *semipredicate* problem, which occurs when a function fails to execute correctly but returns a valid value, thus leaving the caller unaware that an error occurred. Furthermore, exception mechanisms give the programmer an efficient error notification instrument, allow better recovery strategies based on the rich error data available on the exception objects, and allow the programmer to deal with abnormal situations in a civilized way. Nonetheless, and despite the mechanism's broadly recognized qualities on handling and recovering from errors, on our work we show that programmers are not using exception handling constructs as a recovery mean. Most times, when an error occurs, exceptions are silenced or just used to terminate a program in an orderly fashion, not really to recover. We show that the strategies for dealing with exceptions on non-critical programs are commonly non-existent or serve the final purpose of keeping track of problems for later analysis (debugging). Very little effort is normally spent trying to understand exceptions, their causes, and planning recovery actions. As a result, the amount of code found in these applications that is exclusively dedicated to exception handling is usually reduced. This is an unexpected fact. We would anticipate a much larger chunk of code dedicated to exception handling if we consider that: a) Simple operations, such as accessing a file on disk or sending a query to a database, can raise a large number of different exceptions; b) Each different exception type can have several distinct handling actions that may vary with location and time; c) Code for handling an exception can be as or more complex as the code raising the exception; d) In some programming languages (e.g., Java) it is mandatory to handle exceptions and declare their existence.

The unwillingness of software designers to correctly deal with exceptions and follow some well known best-practices for exception handling contributes to the lowering of the quality of programs and their resilience to errors. The premise for our work is that something is not right with current exception handling models: they are not adequate enough for developers. The problem is even more worrisome if we consider that programming

languages designers often neglect the exception mechanism and look at it more like an add-on for their language instead of a central part. As a consequence, software quality suffers as programmers feel that the task of writing good error handling code is too complex, unattractive and inefficient.

In this dissertation we propose a new model that automates the handling of exceptions by the runtime platform. The Automatic Exception Handling (AEH) model frees the programmer from having to write exception handling code and, at the same time, successfully increases the resilience of programs to abnormal situations. The case for automatic exception handling is that, for the majority of situations, benign recovery blocks of code should be part of the runtime platform and should be automatically executed when exceptions are raised. By doing so, the programmer is freed from the “burden” of writing exception handling code for a large number of situations.

The proposed model is influenced by three fundamental concepts: *Exception Handling*; *Software Transactional Memory (STM)*; and *Recovery Blocks*. We incorporate many concepts from traditional exception handling models in order to keep the essential features already available. But, in broader terms, we allow the user to define *handler-free* try blocks, while, at the same time, we set a transactional environment for the execution of these blocks and *system-defined* recovery blocks. Transactions are essential to our model since they provide for atomicity during the execution of protected code blocks and multiple recovery actions. Furthermore, they provide a simple and transparent way of eliminating the effects of failed recovery blocks executions. Our model guarantees that there are no collateral effects arising from the execution of multiple recovery blocks, when it is necessary to execute several of these blocks in order to handle an exception that is repetitively being raised inside a protected block. On the other hand, not all recovery has to be done automatically. The programmer may still deal with a situation on his own, if he or she wishes to do so.

We conclude this thesis by describing and testing an implementation of the proposed model. The results of our experiments are very promising. We obtained a substantial decrease on the amount of exception handling that has to be coded (less 30%), the reliability of programs was improved, and the performance penalty was negligible. Moreover, we were able to propose recovery actions for more than 60% of the exception types we analyzed. We show that the automatic exception handling model: a) can be implemented and incorporated onto existent platforms; b) is easily and almost transparently integrated with object-oriented languages; c) is effective on reducing the amount of exception handling code that programmers have to produce; d) has the

potential to avoid some programming bad practices in terms of reliability and improve the quality of code; e) can reduce development time; and, f) effectively increases the resilience of a system in the presence of errors.

Resumo

Os primeiros mecanismos de detecção e tratamento de exceções surgiram há quatro décadas. Curiosamente, apesar dos mecanismos actuais não serem muito diferentes dos primeiros modelos, a grande maioria das linguagens de programação modernas confia nestes para lidar com erros e situações anormais. As exceções apresentam múltiplas vantagens quando comparadas com outros mecanismos de detecção e tratamento de erros, como por exemplo, a utilização de *códigos de erro* e *variáveis de estado globais*. As exceções eliminam o problema dos *semi-predicados*, caracterizado por funções que mesmo executando incorrectamente devolvem um valor válido, impossibilitando assim a detecção do problema. O mecanismo de exceções dá ao programador os meios para comunicar e detectar a ocorrência de situações anormais, permite a definição de estratégias de recuperação mais elaboradas (com base na informação existente nos objectos que representam exceções), e permite ao programador lidar com situações anormais de uma forma civilizada. No entanto, e apesar das reconhecidas qualidades do mecanismo na detecção e tratamento de situações anormais, os programadores não estão a utilizar os mecanismos de tratamento de exceções como ferramentas para recuperar o estado dos programas após a ocorrência de um de erro. Na maioria dos casos, quando ocorre um erro, as exceções são *silenciadas* ou utilizadas apenas para terminar o programa de uma forma ordenada.

Neste trabalho, mostramos que as estratégias para tratamento de exceções em sistemas *não críticos* não existem ou servem apenas o propósito final de manter registo dos problemas para posterior análise (depuração ou "*debugging*".) Os programadores dedicam pouco tempo a tentar compreender as exceções, a sua origem e a planear métodos de recuperação. Como resultado, a percentagem de código nestas aplicações exclusivamente dedicado ao tratamento de exceções é muito reduzida. Este facto é inesperado se considerarmos que: a) até mesmo operações simples, como aceder a um ficheiro em disco ou executar uma pesquisa numa base de dados, podem originar um grande número de exceções; b) o mesmo tipo de excepção pode requerer diferentes tratamentos e estes podem variar com o momento ou localização do evento anormal; c) o código para tratamento de uma excepção pode ser tão ou ainda mais complexo que o código que originou a excepção; d) em algumas linguagens de programação (e.g., Java) o tratamento de exceções ou a declaração da sua existência é obrigatório.

Este comportamento negligente, por parte dos programadores, parece evidenciar que os mecanismos de tratamento de excepções actuais não se adequam ao perfil dos seus utilizadores. Os programadores consideram a escrita de código de tratamento de excepções uma tarefa complexa, ineficiente e pouco atraente. Esta situação é ainda mais preocupante, tendo em conta que, normalmente, os projectistas de linguagens de programação olham para os mecanismos de tratamento de excepções como um componente periférico ao seu sistema e não como uma parte central. Consequentemente, a qualidade do software irá sofrer.

Nesta dissertação propomos um novo modelo de tratamento de excepções que automatiza o tratamento de situações anormais e o torna numa responsabilidade da plataforma de execução. O nosso modelo liberta o programador da tarefa de escrever código para lidar com erros ou situações inesperadas e, simultaneamente, aumenta a resiliência dos programas aos erros. O modelo de Tratamento Automático de Excepções (TAE) proporciona uma forma efectiva de lidar com excepções sem interferir com a produtividade dos programadores. Para um grande número de excepções é possível que a própria plataforma de execução forneça, aquando de uma excepção, blocos de código de recuperação benignos capazes de recuperar o estado de um programa e permitir a continuação da execução. Desta forma, os programadores ficam livres da tarefa de escrever código de tratamento para um grande número de excepções.

O modelo proposto tem por base três fontes de influência muito díspares: os modelos de *tratamento de excepções* existentes; o mecanismo de *blocos de recuperação*; e o mecanismo de *memória transaccional por software*. De forma a preservar algumas funcionalidades já existentes, incorporámos no nosso modelo muitos conceitos associados aos mecanismos de tratamento de excepções mais eficientes. No entanto, permitimos que o programador defina blocos `try{}` sem ter de criar blocos de tratamento associados (e.g., blocos `catch` ou `finally`) e, simultaneamente, definimos um ambiente de execução transaccional para esses blocos `try` e para os blocos de recuperação de excepções automáticos implementados ao nível do sistema. As transacções são essenciais ao nosso modelo, sendo que, garantem a atomicidade da execução dos blocos `try{}` e dos múltiplos blocos de recuperação. Mais importante ainda, é o facto das transacções permitirem eliminar de uma forma limpa e transparente, os efeitos da execução de blocos de código onde ocorreram excepções.

O ambiente de execução transaccional assume uma importância ainda maior quando constatamos que a forma de recuperar de uma excepção, apesar de ser correcta numa

situação, pode mostrar-se totalmente inadequada noutra, para o mesmo tipo de excepção. Assim, em certas ocasiões, será necessário experimentar diferentes tipos de tratamento antes de o sistema conseguir recuperar da excepção. O nosso modelo assegura que quando é necessário executar mais do que um bloco de recuperação, de forma a eliminar uma excepção reincidente dentro do bloco de código protegido, não irão existir efeitos colaterais da execução dos vários blocos de recuperação ou das várias tentativas de execução do bloco de código protegido. Por outro lado, o tratamento de excepções não tem de ser totalmente automático. O programador pode optar por ser ele a definir o tratamento para uma ocorrência excepcional específica, se assim o preferir.

Concluimos esta dissertação com a discussão, teste e validação de uma implementação do modelo proposto. Os nossos testes mostram que é possível obter uma redução substancial na quantidade de código de tratamento de excepções que é necessário escrever (menos 30%). Também é perceptível um aumento assinalável na resiliência aos erros dos programas analisados. Além destas melhorias, foi possível observar que o impacto da utilização do novo modelo na performance dos sistemas estudados pode ser considerado negligenciável. Finalmente, para provar que é exequível a criação de acções para recuperação automática de excepções ao nível do sistema, desenvolvemos um conjunto acções benignas capazes de lidar com mais de 60% dos tipos de excepções analisados. No geral, mostramos que o modelo de tratamento automático de excepções: a) é passível de ser implementado e incorporado em plataformas de execução e desenvolvimento já existentes; b) é facilmente e quase de forma transparente integrável com linguagens de programação *orientadas-aos-objects*; c) é eficaz a reduzir a quantidade de código dedicada ao tratamento de excepções que os programadores têm de escrever; d) tem potencial para melhorar a qualidade do código final das aplicações e evitar que o programador “caia” em más práticas de programação no que diz respeito à robustez dos programas; e) pode reduzir os tempos de desenvolvimento; e, por último, f) aumentar globalmente a robustez de um sistema.

Acknowledgments

The conclusion of this work is, without any doubt, among the most important moments of my life. Therefore, I can not let it pass without thanking everyone that accompanied me through these years and that supported me unconditionally.

I would like to start by thanking my advisor, Professor Paulo Marques, without whom I would not be concluding this work. He was responsible for opening to me the doors of this “curious” world of research. First, when he invited me to work as developer on the RAIL project, and later when he challenged me to do this PhD. But, above all, I want to thank him the privilege that was for me to work and learn with someone that lives his work with such an intense way and that shows an energy, knowledge, wisdom, dedication, and enthusiasm hard to beat.

I also wish to thank Professor Luís Silva for taking me as his student on the early stages of this dissertation and for the trust that he always put on my work. I am also truly grateful to Paulo Sacramento and Hugo Matos for their contribution to this dissertation. Both developed part of the software used on the studies and tests that we performed. Paulo also conducted his graduation work with me and co-authored two articles. I also want to acknowledge Patrício Domingues, my lab colleague, which with whom was a joy to work!

Finally, I want to thank my lovely wife Rita for making me happy. I am also deeply thankful to my parents, my sister and all my family for their amazing support and care. I also wish to thank my friends, whose contributions took many different forms and were more significant than they probably realize - Dorita, Granjal, Barreto, Ana, Joana, Sebastião, Lúcia, Dulce, Jorge, Cláudio Jorge and Ana Pinto, I thank you all.

Table of Contents

ABSTRACT	IX
RESUMO	XII
ACKNOWLEDGMENTS	XV
TABLE OF CONTENTS	XVII
LIST OF FIGURES	XIX
LIST OF TABLES	XXI
LIST OF CODE SAMPLES	XXIII
1. INTRODUCTION	1
1.1. Motivation	2
1.1.1. A first glance at exception handling mechanisms	4
1.1.2. Exception Handling Design Issues	9
1.2. Research Objectives	21
1.3. Contributions	22
1.4. Structure of the Dissertation	22
2. CURRENT APPROACHES TO EXCEPTION HANDLING	25
2.1. Introduction	26
2.1.1. First efforts in the definition of a standard notation	27
2.2. Handling models: features and propagation	30
2.2.1. Handling models	31
2.2.2. Features	33
2.3. Evaluation and quality metrics	52
2.3.1. Evaluation	53
2.3.2. Quality requirements	59
2.4. Backward error recovery	62
2.5. Real-time concerns	65
2.6. Other approaches	66
2.6.1. Aspect Oriented Programming	67
2.6.2. Exception handling for Futures	68
2.6.3. Compensation stacks	70
2.7. Summary	73
3. A FIELD STUDY IN EXCEPTION HANDLING	77
3.1. Introduction	78
3.2. Programming with exceptions	81
3.2.1. Methodology	81
3.2.2. Results	86

- 3.2.3. Related work108
- 3.3. Documenting exceptions113**
 - 3.3.1. Motivation113
 - 3.3.2. Methodology and Tools115
 - 3.3.3. Results120
- 3.4. Summary.....126**

- 4. AUTOMATIC EXCEPTION HANDLING: A PROPOSAL129**
 - 4.1. Introduction130**
 - 4.2. The Model131**
 - 4.2.1. Benign Recovery Actions133
 - 4.2.2. Programming Model.....136
 - 4.2.3. Transactional System138
 - 4.2.4. Exception Parameters142
 - 4.2.5. Exception Handling Model Features144
 - 4.3. Related Work149**
 - 4.4. Summary.....153**

- 5. IMPLEMENTATION AND VALIDATION.....155**
 - 5.1. Introduction156**
 - 5.2. Framework implementation.....156**
 - 5.2.1. The STM Library.....161
 - 5.2.2. The AEH Class Loader164
 - 5.3. Validation and Testing.....171**
 - 5.3.1. Source Code.....172
 - 5.3.2. Resilience174
 - 5.3.3. Recovery Actions.....180
 - 5.4. The Perfect Exception Handling Model186**
 - 5.5. Summary.....188**

- 6. CONCLUSION189**
 - 6.1. Overview189**
 - 6.2. Contributions189**
 - 6.3. Future Work.....189**

- LIST OF PUBLICATIONS189**

- BIBLIOGRAPHY189**

List of Figures

Figure 1.1 – Example of source code in Java using exception handling constructs	5
Figure 1.2 – A simplified flowchart for exception propagation outside <code>try{}</code> blocks.....	8
Figure 1.3 – A simplified flowchart for exception propagation from inside a <code>try{}</code> block .	9
Figure 2.1 – Label variables usage example	30
Figure 2.2 – Java code exemplifying the termination model	31
Figure 2.3 – Retry model exemplified with Eiffel notation.....	32
Figure 2.4 – Resumption model exemplified with Smalltalk notation	33
Figure 2.5 – Coordinated Atomic Actions scheme overview	51
Figure 3.1 – Amount of exception handling code	87
Figure 3.2 – <code>Catch</code> handler actions for .NET programs.	90
Figure 3.3 – <code>Catch</code> handler actions for Java programs.....	92
Figure 3.4 – Count of actions for <code>Finally</code> handlers in .NET programs.	93
Figure 3.5 – Count of actions for <code>Finally</code> handlers in Java programs.....	94
Figure 3.6 – .NET classes being used as catch arguments.....	96
Figure 3.7 – Java classes being used as catch arguments.	97
Figure 3.8 – Handler actions distribution for the most used catch handler classes.	98
Figure 3.9 – Most commonly handled exception types in .NET.....	102
Figure 3.10 – Most commonly handled exception types in Java.....	103
Figure 3.11 – Call stack levels for caught exceptions.....	104
Figure 3.12 – Handlers size in number of IL code instructions for .NET.....	105
Figure 3.13 – Number of catch handlers per protected block.	107
Figure 3.14 –Automatic documentation of an exception using specialized tags.	114
Figure 3.15 – Dictionary: IL instruction/opcode/list of exceptions.....	117
Figure 3.16 – Scheme of the code analysis process.....	118
Figure 3.17 – Documentation of exceptions in four different assemblies.	123
Figure 4.1 – The runtime system provides recovery.....	138
Figure 4.2 – Passing parameters to recovery blocks.	142
Figure 5.1 – AEH system architecture.	158
Figure 5.2 – Loading and running applications using the AEH.....	159
Figure 5.3 – Eclipse plug-in	170
Figure 5.4 – Configuration interface.....	171
Figure 5.5 – Testing framework.....	174
Figure 5.6 – Analysis of the executions times.	178
Figure 5.7 – Description of the experience.....	179
Figure 5.8 – JMeter workload run summary.....	180

List of Tables

Table 1.1 - Exception handling best practices.....	14
Table 1.2 - Exception handling antipatterns.....	16
Table 1.3 - Exception handling and the object-oriented paradigm.....	19
Table 2.1 - Identification of the exception handling models evaluation items.....	53
Table 3.1 - Applications listed by group.....	83
Table 3.2 - List of Assemblies and Java Packages analyzed.....	85
Table 3.3 - Description of the Handler's actions categories.....	89
Table 3.4 - Java and .NET exception classes for bytecode and IL code instructions.....	101
Table 3.5 - Number of protected blocks, catch handlers and finally handlers.....	106
Table 3.6 - Usage of Unchecked exceptions in Java catch handlers.....	107
Table 3.7 - Group Characterization.....	119
Table 3.8 - Assemblies used in the study.....	119
Table 3.9 - Documented vs. Undocumented exceptions.....	121
Table 3.10 - Types of exceptions most likely to be documented.....	122
Table 3.11 - Suspects for all eight Assemblies.....	124
Table 3.12 - Type of detections responsible for code suspects.....	125
Table 3.13 - Proportion of detections due to lack of documentation.....	125
Table 5.1 - Causes for <code>JMSEException</code> to be raised.....	173
Table 5.2 - Exception Injection Results (all apps.).....	175
Table 5.3 - Results with content checking (3 apps.).....	176
Table 5.4 - System availability, MTTR, MTBF and error rate.....	180
Table 5.5 - Recovery actions for Java's <code>IOException</code> class tree.....	181
Table 5.6 - Applications source code decrease analysis.....	184
Table 5.7 - Healing strategies.....	185
Table 5.8 - Exception model features list.....	187

List of Code Samples

Listing 1.1 - Sample from <i>LimeWire</i>	11
Listing 1.2 - Writing to a file	12
Listing 2.1 - Multiple derivation for derived exceptions	34
Listing 2.2 - Bound exceptions and conditional handling	35
Listing 2.3 - Dynamic propagation through an invisible scope.	40
Listing 2.4 - Recursive resuming example	41
Listing 2.5 - Handler's static context in C++ and Ada	43
Listing 2.6 - The notation for a recovery blocks structure	63
Listing 2.7 - Futures utilization within the <i>DBLFutures</i> framework	69
Listing 2.8 - Examples of exception handling in <i>DBLFutures</i>	70
Listing 4.1 - Writing to a file in a transactional <code>try</code> block	132
Listing 4.2 - <code>swap</code> method using an <code>atomic</code> block and alternative execution paths.	149
Listing 4.3 - Implicit and Explicit reconstructors declaration	151
Listing 4.4 - Context management integration in <code>try/catch</code> blocks	152
Listing 5.1 - The <code>Transaction</code> class.	160
Listing 5.2 - The <code>commit()</code> method	162
Listing 5.3 - The <code>ITransObject</code> interface	162
Listing 5.4 - The <code>canCommit()</code> method	163
Listing 5.5 - The <code>doFinalCommitTasks()</code> method	164
Listing 5.6 - The <code>TransactionClassAdapter()</code> methods	166
Listing 5.7 - Recovery actions for the <code>JMSException</code> class.	168
Listing 5.8 - Configuration file example	169

Introduction

This thesis is the result of research done in exception handling mechanisms for object-oriented programming languages at the Software and Systems Engineering Group of the University of Coimbra, Portugal.

In this opening chapter, the motivation and research objectives of the investigation are described, providing a foundation for the upcoming discussion. Finally, a brief summary of the contributions of the dissertation is presented.

1.1. Motivation

For more than one hundred years, since the time when the word computer still referred to the “person who performed computation” and not a machine, programmers have been worried about faults and errors that they may raise in calculations [Randell1982]. These faults can have many different natures and causes. They can be accidental or intentional, malicious or not-malicious, physical (hardware) or human. The main concern is that, independently of their nature, they can cause errors [Laprie1995b].

To avoid these errors many approaches have been taken, most of them at hardware level. These included the widespread use of error detecting and correcting codes¹, the use of replicated² processors, voting³, masking⁴ and automatic reconfiguration. Hardware fault tolerance schemes [Randell1978] try to be as simple as possible to avoid expensive trade offs like lost of performance and computation power, increased cost and energy consumption. Nevertheless, even if we could eliminate hardware faults, residual software design faults would continue to affect programs. Therefore, extra care had to be taken at software level to prevent or handle these faults in the quest to avoid errors. Some of these solutions mingled hardware and software approaches such as fault avoidance, redundancy, masking and reconfiguration. Others, to ensure better portability and avoid the need for dedicated hardware, preferred a software-only approach.

One of the most popular software programming languages mechanisms for dealing with abnormal behaviors is *exception handling*. Since the seminal work of John B. Goodenough [Goodenough1975] in the definition of a notation for exception handling and Flaviu Cristian [Cristian1980] in defining its usage, the programming language constructs for handling and recovering from exceptions have not changed much. Exception handling is basically a civilized way of dealing with exceptional situations (occurrence of a condition that changes the normal flow of execution of a program). It represents a significant improvement over other error handling mechanisms like checking *return codes*, additional *boolean state flags*, among others. For instance, exception handling eliminates the

¹ A method and apparatus for detecting and correcting bit errors in data streams [Hamming1950].

² Replication is based on the usage of multiple instances of the same system that are able to execute the same function in parallel.

³ Mechanism used in systems with replicated components (3 or more) to mask the faulty components.

⁴ Masking is used to manipulate the effects of faults in order to ensure that systems always behave as specified and, hence, users always observe the expected behavior [Jalote1994].

Semipredicate Problem that occurs when a function fails to execute correctly but returns a valid value, thus leaving the caller unaware that an error occurred. Furthermore, exceptions give the programmer an efficient error notification mechanism, allow better recovery strategies based on the rich error data available on the exception objects, and improve readability by separating exception handling code from the business logic code. Exceptions introduce their own error handling flow-of-control to the program. When an exception is raised, the execution flow is diverted to the appropriate error handling code.

Exceptions can have three different origins [Doshy2003]:

1. *Programming errors*: exceptions can be raised due to programming errors, such as accessing *null* references. The code which invokes the function raising the exception (client code) cannot do anything about programming errors;
2. *Client code errors*: exceptions can be raised when the calling code attempts to perform some operation not allowed by the API, thus violating the contract. If the exception provides enough information, the client code can try an alternative path;
3. *Resource failures*: exceptions can be raised to acknowledge resource failures, such as when the system runs out memory or a network connection fails. The client code response is context-driven. The operation could be retried after some time or the application halted.

If we consider that there are an infinite number of computations that one can think of, there will also be an infinite number of reasons for a program to raise an exception. For instance, a program may: try to access an *out-of-bounds* array element; try to access members on a null reference; perform an integer “divide by zero” operation; try to unsuccessfully parse a string to an integer; try to open a file that does not exist; or get an IO error. Depending on the exception being raised, the location and the moment (where and when it is raised) in the program, the cost of mishandling such an event can be high – e.g., “not handling a failure on a withdraw operation on an ATM machine may lead to an improper decrement of the client’s account balance if he or she does not receives the requested amount”.

1.1.1. A first glance at exception handling mechanisms

Exception handling models and their implementations vary from programming language to programming language¹. But, in general, when an abnormal situation is detected the program raises an exception. When an exception is raised inside a protected region of code (guarded code block), the execution flow is transferred to a predefined location known as the exception handler. Usually, the stack is unwound and the extra-information necessary to handle the exception, such as its name, description, location, and severity is communicated either by the code that has raised the exception, or by the exception handling supporting mechanism. The handler code determines what happens next: the program may try to recover from the exception; just log its occurrence and terminate the application in an orderly fashion; or simply ignore the exception. In some cases, after the execution of the handler, it may be possible to resume the execution of the program at the original location and reset the program's state prior to the exception occurrence². Exception handlers can be associated with exception types, classes, methods, objects, or blocks of code. Handlers that remain valid in any part of an application are called *default handlers*. When an exception is raised, the *normal* flow of execution of the application is deviated allowing the system to search for a suitable handler. The execution returns to the normal flow immediately after the invocation of the selected handler (when found). The point where the normal flow of execution is to be resumed depends of the model for the continuation (termination/resumption model). If no suitable handler is found, the execution of the program is terminated.

The example in Figure 1.1 illustrates the usage of the exception handling constructs in an object-oriented programming language - Java [Gosling2005]. This piece of source code is useful to help understand the concepts and the execution flow issues associated with nowadays exception handling models. Although existent exception handling models do not differ much, there are still some important differences between implementations. This example (and its subsequent analysis) does not intend to address or represent all the

¹ Chapter 2 provides a thorough discussion on exception handling models.

² This behavior, associated with the resumption model [Goodenough1975], is not supported in all platforms and programming languages. The most widely used approach is the termination model [Goodenough1975]. In the termination model, control flow after the handler execution continues as if the failed instruction in the protected block is terminated without encountering the exception.

<pre>void foo() { //doing something outside a protected block</pre>	A
<pre> try { //doing something inside a protected block }</pre>	B
<pre> catch (IOException e1) { //handling an IO exception }</pre>	C
<pre> catch (Exception e3) { //handling other exception type }</pre>	D
<pre> finally { //performing final computations }</pre>	E
<pre> //doing something outside a protected block }</pre>	F

Figure 1.1 - Example of source code in Java using exception handling constructs

models available in modern programming languages¹. It is merely an illustration of how things can work, how the source code is organized (when using exception handling constructs), and how the program behaves in the presence of exceptions.

In the example of Figure 1.1, the source code for method `foo()` is divided in six different parts. Parts *A* and *F* represent the first and the last instructions on the method. Part *B* is bordered by a `try{}` block, it corresponds to a protected region of code where the programmer knows that some exceptions are prone to happen. Parts *C* and *D* are the handlers for the exceptions being raised in part *B*. The `catch` instruction is used to mark the beginning of the handler block (that is limited by braces) and accepts, as a parameter, the name of the class for the exception to handle. For instance, the catch handler in part *C* deals with all the exceptions raised in *B* that are instances of (or descend from) the `IOException` class, and the catch handler in part *D* handles the remaining exception types derived from the `Exception` class. Part *E* represents a special block of code (`finally{}`)

¹ The Java Programming Language was chosen for this and other examples on this document because it is a well known mainstream programming language with a state-of-the-art exception handling mechanism.

that will always execute, independently of an exception being raised or not inside parts *B*, *C* and *D*.

Checked and unchecked exceptions

Exception handling is more than just `try-catch-finally` blocks. It also encompasses two important aspects, related between them. One is the conceptual relation between a method and the exceptions it can throw; the other is the existence of an obligation to handle the exceptions that are thrown by a method. These aspects define the essence of two different exception models, the *checked exceptions model* and the *unchecked exceptions model* [Gosling2005].

For instance, the Java programming language designers (among others) believe that certain exceptions impact the functionality of a method so intrinsically that they should be explicitly declared, being the programmer forced to handle them – thus justifying the option of implementing the checked exceptions model. On the other hand, other designers [ISO23271:2006] believe that the programmer should not be forced to handle all the exceptions. In Java, for instance, a small set of exceptions (runtime exceptions) are explicitly marked as unchecked. The programmer is free to choose which exceptions he or she wants to explicitly deal with and which prefers not to. Checked exceptions can interfere with the programmers' productivity, since they cannot concentrate in business logic and are constantly forced to think about errors. Furthermore, .NET [ISO23271:2006] creators (in particular) advocate that errors should be "exonerated" by exhaustive testing. I.e., a sufficiently accurate test suite should be able to expose dormant exceptions, and corresponding abnormal situations. For the problems that remain latent, it is better that they appear as a clean exception that terminates the application rather than having them being swallowed in a generic catch statement which can lead to a corrupt state.

In the checked exception model, programmers have to declare the exceptions that a method throws. For this purpose, the programming language provides the constructs necessary to create a list of exceptions in the method's signature. As a consequence, code invoking methods with a declared exception list has only two options:

- Handle the declared exceptions with a `try-catch-finally` structure;
- Declare to propagate the same set of exceptions that the invoked method does.

Failing to comply with the previous rules will result in a compile error. This prevents a known problem from being propagated throughout the program or remaining unhandled on the final software product.

Systems that implement the unchecked exception model are not able to provide this kind of safeguard. The developer is able to declare the exceptions that a method throws, but he or she is not forced to handle any exceptions. The compiler will never emit warnings about unhandled exceptions thus an error, which otherwise would not be fatal, can be unstopably propagated on the call stack and cause the program termination. The only way to detect this kind of problem is through code analysis or exhaustive testing. To know what exceptions an operation may raise, the developer has to trust the documentation. To simplify the documentation task, some languages provide meta-tags allowing the generation of automatic documentation.

Explicitly raising an exception gives the programmer the opportunity to fill the exception *object* with all the relevant information necessary for dealing with the abnormal occurrence at hands. Exception types can be *user-defined* or *predefined*. User-defined exceptions are created and detected at application level. Predefined exceptions are built-in into the runtime platform libraries and detected by the runtime platform. Consider the following example: a *program tries to open a file that does not exist* – in these circumstances the program will raise an exception. But, if the programmer wishes to prepare the application to recover from this exception, he or she needs more information:

1. The correct classification of the error – a well designed exception will transmit this information merely by its type (e.g., the class `FileNotFoundException` is self explanatory);
2. The path for the missing file;
3. The name of the file.

If this information is attached to the exception at the time of its generation, the programmer can use it to design a handler that, for instance, will look for the file and open it in a different location – local folder, network folder, URL, etc. Thus, if successful in opening the file, the program is able to continue the computation in a valid and clean state.

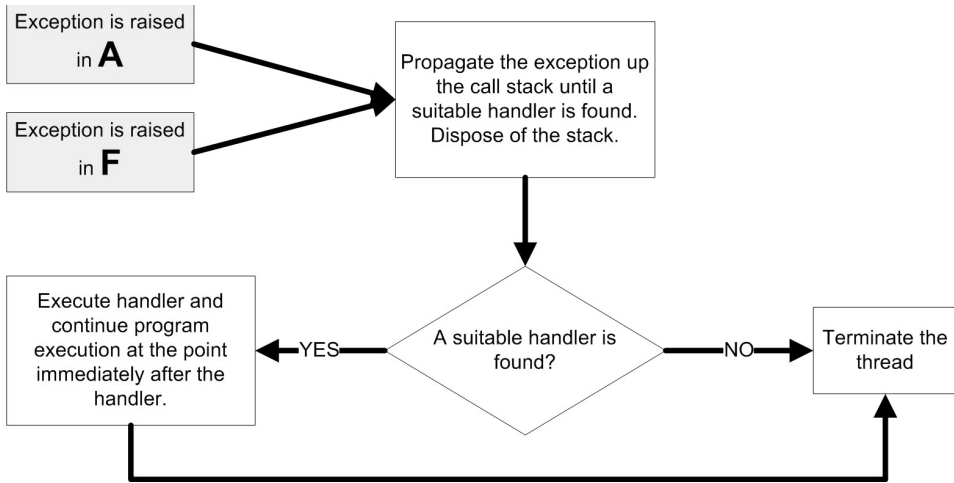


Figure 1.2 – A simplified flowchart for exception propagation outside try{} blocks

Execution flow and exception propagation

A fundamental issue with exception handling mechanisms is what happens when an exception is raised: a) how is the execution flow affected; b) what happens to the stack information; c) how is the correct handler for a particular exception elected; and d) where will the execution continue after the completion of the handling actions. Many of these aspects will vary with the chosen programming language and its corresponding exception handling model¹. For the purpose of this section, an explanatory example will give a first insight on how exception flow can be driven inside a running application.

The flowchart in Figure 1.2 describes the propagation of an exception raised inside parts *A* or *F* of the code on Figure 1.1. This exception must be an unchecked exception. Otherwise, as it has been described before, the compiler would have complained and aborted the compilation (because the method does not declare the exceptions that it may raise). In the code example, one can observe that parts *A* and *F* are not protected regions of code since they are not inside a try{} block. Thus, when the exception is raised, the runtime aborts the execution of the method foo(), unwinds the stack, and propagates the exception up the call stack to foo()'s caller. This is done in order to look for a suitable handler. In Java, a handler is considered suitable if it catches the class of the exception or a parent class of the

¹ Once more, Chapter 2 provides a thorough discussion about different exception handling models implementations.

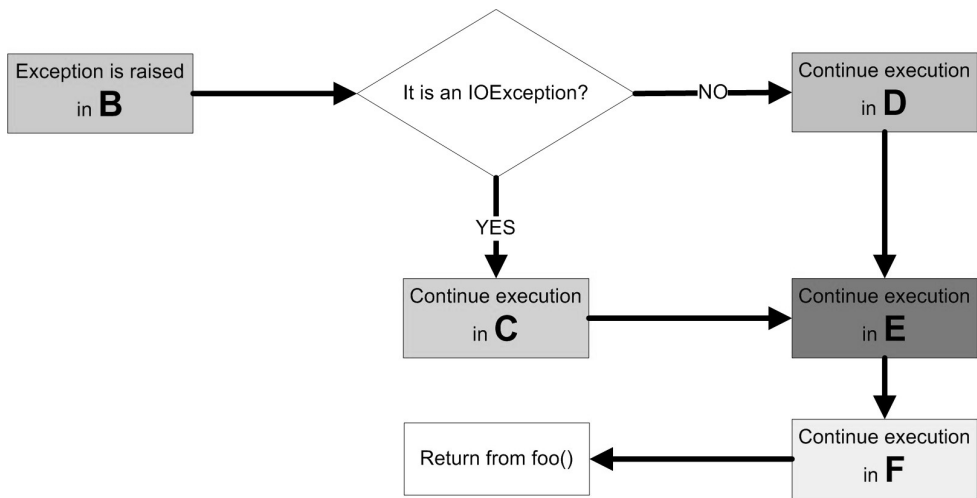


Figure 1.3 – A simplified flowchart for exception propagation from inside a `try{}` block

exception class. If a handler is selected for execution, the program *normal* execution continues in the instruction that immediately follows the handler finale. If no handler is found, the runtime repeats the process and continues to unwind the stack, propagating the exception, until the corresponding thread terminates.

The flowchart in Figure 1.3 illustrates a different scenario. In this example the exception is raised inside the `try{}` block (*B*). Afterwards, depending on the class of the exception, the execution will continue in the first instruction of the `IOException` class handler (*C*) or of the `Exception` class handler (*D*). Independently of which of the handlers is executed, the first instruction on *E* (`finally` block) is always executed next. If no other exception is raised, the flow of execution will continue through *F* until the method's return point. In the event that no exception occurs in *B*, or that an exception occurs in *C* or *D*, the code in part *B* will always be executed but, in the later case, the method returns immediately and the exception is propagated up the call stack.

1.1.2. Exception Handling Design Issues

Exception handling mechanisms represent an improvement over traditional error handling mechanisms. They introduced an organized, reliable, focused and self-explanatory way to deal with errors and abnormal situations. But, besides the obvious benefits, exception handling is far from perfect. In fact, it can be argued that the mechanism is seriously

limited if not even flawed as a programming construct. Most times, when an error occurs, exceptions are silenced or just used to terminate a program in an orderly fashion, not really to recover.

In most cases, strategies to deal with exceptions are non-existent or just serve the final purpose of keeping track of problems for later analysis. Very little effort is normally spent trying to understand exceptions, their causes, and planning recovery actions. Christian [Christian1995] argued that more than two thirds of a program's code is often devoted to detecting and handling errors and exceptions, and, according to Utas [Utas2004], three quarters of the code are dedicated to error and exception handling. An important part of our work was devoted to identifying exception handling patterns and exception programming problems in modern software¹. We now know that most of the software running in our computers, servers, networks, at home or at work, is prone to have less than 10% of code dedicated to exception handling and the most common percentage is around 5% [Cabral2007]. Our conclusions are supported by Shah's recent work [Shah2008a] on explaining why "developers neglect exception handling code". This is an unexpected fact. We would anticipate a much larger chunk of code dedicated to exception handling if we consider that:

1. Simple operations, such as accessing a file on disk or sending a query to a database, can raise a large number of different exceptions;
2. Each different exception type can have several distinct handling actions that may vary with location and time;
3. Code for handling an exception can be as or more complex as the code raising the exception;
4. In some programming languages (e.g., Java) it is mandatory to handle exceptions and declare their existence.

¹ Chapter 3 presents a field study on exception handling

```
try
{
    _socket.send(_dp);
}
catch(ConnectException ce)
{
    // oh well, can't connect, ignore it...
}
catch(BindException be)
{
    // oh well, if we can't bind our socket, ignore it..
}
catch(NoRouteToHostException nrthe)
{
    // oh well, if we can't find that host, ignore it ...
}
catch(IOException ioe)
{
    if(isIgnoreable(ioe, ioe.getMessage()))
        return;

    String errString = "ip/port: " +
        _dp.getAddress() + ":" +
        _dp.getPort();
    _err.error(ioe, errString);
}
```

Listing 1.1 – Sample from *LimeWire*

More important than the amount of code dedicated to exception handling is the quality of the code and its ability to recover the program to a valid state. Writing good exception handling code is complex, cumbersome and error-prone task. As an example, the Java code block in Listing 1.1 is a small portion of the source code of the *LimeWire* client for the *Gnutella* peer-to-peer network [Limewire2009]. This widely used piece of code is responsible for the sending of UDP packets in the application. The example shows that the single operation of sending a `DatagramPacket` over the network is prone to raise several exceptions. In ideal conditions, the programmer may have to provide a different handler for each exception. This means that a different catch block is expected for each different exception type being raised. In theory, these handlers could implement the code necessary to recover from the raised exception and correctly complete the method's execution. Unfortunately, providing the recovery code is a complex error prone task and, in consequence, programmers prefer to ignore the exceptions or deal with them later (even if the stack and other useful information are no longer available). Furthermore, the code for handling the raised exceptions may originate other exceptions by itself. This can lead to possible undesired sceneries where protected blocks (`try` blocks) are nested inside other protected blocks, introducing a great level of tangling in the flow of execution of the program. This escalating complexity is certainly influencing the programmers to keep their exception handling code as simple as possible or, in many cases, completely inexistent.

```
// The FileWriter must be declared outside of the try block
// and be pointing to something (null is a common choice)
FileWriter file = null;

try
{
    // open file
    file = new FileWriter("data.txt");

    // write some data into it
    for (int i=0; i<1024; i++)
        file.write("Here's the data: " + i);
}
catch (FileNotFoundException e)
{
    // Deal with filename problems
}
catch (SecurityException e)
{
    // Deal with problems like wrong permissions
}
catch (IOException e)
{
    // Deal with other (generic?) I/O problems
    // How do I do this???
}
finally
{
    try
    {
        file.close();
    }
    catch (IOException e)
    {
        // what should I do???
    }
}
```

Listing 1.2 - Writing to a file

Listing 1.2 helps to illustrate the potential increase in the complexity of code as one adds exception handling. Consider that the programmer just wants to write some data into a file. The data itself is not of much importance, just the fact that the programmer wants to correctly save it to disk. From a “core” algorithmic perspective, the intent of the programmer is to write a couple of lines similar to the ones shown with the grey shadowing in the example. Now consider what happens on a language that uses checked exceptions, like Java, or even a language that does not have checked exceptions, like C#, but on which the programmer wants to correctly deal with possible abnormal situations. (After all, if one wants to develop robust software, exception handling is necessary.) The programmer ends up having to write code for dealing with exceptions like `FileNotFoundException`, `SecurityException`, `DiskFullException`, just to name a few, or other I/O-related problems (`IOException`).

Already a problem can be seen. The programmer is trying to write some data into a file. But, in order to do so, it is being forced to think about all possible problems that can happen in the process. That is not a bad thing *per se*, but is happening at the wrong time. Writing data into a file is most likely part of a larger algorithm, which may itself be complicated. Thus, in most cases, the programmer is concentrating on that application logic and not really on what goes on when an exception is thrown, or exactly what exceptions can be thrown and their associated recovery actions. A common consequence of this mismatch (thinking about application logic vs. thinking about exception handling) is that many programmers actually silence exceptions in order to be able to keep implementing the core application logic. The rationale is that latter on they will deal with the exceptions, which in many cases never happens.

To complicate things further, in the example, an experienced programmer may argue that if an exception occurs, we should probably try to close the file. That is normally done on a `finally` block or inside an exception handling block. But, in order to do so, a valid file reference must exist at that point. Thus, the declaration of `FileWriter` must be performed outside of the `try` block and the variable must actually be initialized to something (`null` is a common choice). (If the variable is not initialized, the compiler would complain that on the `finally` block the variable could have not been initialized due to an exception during the object instantiation.) There is also the problem of what to do if an exception is thrown while trying to close the file inside of the `finally` block. *Silencing*¹ the exception is a common approach.

Comparing the code that the programmer wants to write (shadowed text in Listing 1.2) with the code that the programmer ends up writing (all the code in Listing 1.2), something is clearly wrong with the model. It is what in McConnell’s nomenclature would be classified as a “coding horror” [McConnel2004], greatly contributing to fragile code and substandard quality. While the programmer is just trying to write some data into a file, it is nonetheless being forced to massively deal with error handling. And, in the case of this example we have not even discussed the code that would be included inside of the exception handlers for error recovery. If one considers medium to large scale applications, with possibly hundreds or thousands of exceptions having to be addressed, this is clearly problematic.

¹ Silencing exceptions is a well known bad practice for exception handling. It consists in an empty catch block with no other purpose than avoiding the propagation of an exception occurring inside a try block.

Best practices for exception handling

The software industry and the programming community in general have established a set of design patterns and guidelines for writing better software using exception handling. These best practices are well known in the software industry and have been thoroughly discussed by several authors in recent years. Nevertheless, programmers are not always keen on following such directives as we will discuss next. Table 1.1 summarizes several widespread exception handling best practices [Wirfs-Brock2006,Muller2002,Doshy2003].

Table 1.1 - Exception handling best practices.

Guideline	Description
Key design issue	Exception handling must be taken seriously as key design issue throughout the whole software life cycle. It is important to start collecting exception handling and logging requirements early. The design of the exception hierarchy must be completed before the start of the implementation.
Indicative exception names	Exceptions should be named after what went wrong and not, for instance, who raised it.
Custom exception types should always provide extra and useful information	Creating a new exception type that does not gives any useful information to the client code, other than an indicative exception name, is not useful. Exception type should provide functionality and information that will help treating the abnormal situation.
Do not declare lots of exception types	Only create a new exception type when its occurrence will be handled differently.
Document custom exception types	Exceptions being thrown by a method should always be documented.
Know when to use checked exceptions and when not to	The usage of checked exceptions is not a reason per se for disregarding the usage of unchecked exceptions. If possible, both models should coexist: exceptions that signal an untreatable situation should be unchecked; exceptions that signal an abnormal situation which might be treatable should be checked.
Preserve encapsulation	Implementation-specific exceptions must never escalate to higher layers. Recast lower-level exceptions to higher-level ones when raising the abstraction level.
Providing context	The exception should carry the necessary context information in order to allow an informed response by the handling code.
Handling code must be close to the problem	The longer an exception propagates in the call stack, the more difficult will be for the handling code to make meaningful decisions.
Use decision empowered objects to handle exceptions	Usually the objects better equipped to treat exceptions are the ones closer to the problem, but, some times the most able object is the one that was designed to control actions and make decisions.

Guideline	Description
Chaining exceptions	In some cases, to maintain encapsulation and hide implementation details that are not useful for the user (another program or a person), it is useful to change the class of the exceptions on their way up the call stack. Unfortunately, this practice inevitably leads to the loss of the stack and possible error information available on the original exception object. If the programming language permits it, a good practice is to chain the original exception inside the newly created one, thus keeping all the information available.
Do not use assertions inside catch blocks	Assertions have no place inside catch blocks, it is just too late to use them: assertions do not provide means to chain an original exception. Making an assertion always false is an abuse of the concept, if assertions are disabled, the catch block ends up empty.
Clean up	Always clean up resources like database connections, network connections, and others. If the programming language allows the usage of finally blocks, be sure to use them for the clean up actions.
Never use exceptions for flow control	Exceptions should not be used to control the execution flow of an application, they should only be raised on emergencies. Generating stack traces is expensive and their information is only useful for debugging purposes.
Do not ignore exceptions	If a software designer deliberately declares that a method throws an exception, he is telling the programmer that the exception must not be silenced or ignored but dealt with. If the programmer believes that handling the exception does not make sense, he or she should catch it, convert it to another form, and re-throw it (possibly as an unchecked exception if the model allows it).
Do not catch top-level exceptions	General (top-level) exception types, like Java's <code>java.lang.Exception</code> , do not give much information about the underlying problem. The programmer should catch the exact exceptions types that the invoked functions <i>declare</i> to throw and treat them separately. Putting all the "occurrences in the same bag" is a poor error recovery practice since no detailed information about the problem is available.
Do not try to handle coding errors	The cost of trying to handle coding errors is extraordinary, thus, only high fault-tolerant systems will require such extraordinary measures.
Log only what is important	Log should be done as late as possible and each exception must be logged exactly once. Do not use handlers that only log and repeatedly re-throw the same exception (this is an expensive practice). The best moments to log an exception are when it is: being treated; leaving a physical tier/virtual machine through a remote call; leaving a logical tier with its own log file.

It is interesting that many of the discussions about exception handling programming rules are currently evolving into the discussion of exception handling design patterns, much inspired by the well known object-oriented design patterns [Gamma1995]. The first steps in this direction are already visible at Portland Pattern Repository [Portland2007].

Exception handling antipatterns

No one argues that the described best practices help to increase the overall software quality, but, we can say that *for each good design pattern there is always opposed an antipattern* (this could certainly be the software world equivalent to the Newton's third law of motion). Tim McCune was the first author to introduce the concept of *Exception Handling Antipatterns* [McCune2006], much inspired by the 1998 release of *AntiPatterns: Refactoring Software, Architectures, and Projects in a Crisis* [Brown1998]. The author argues that for many novice to mid-level developers exception handling tends to be an afterthought - "*try-catch-print the stack trace*" is the most common exception handling pattern used by these developers - and, if they attempt to incorporate elaborated schemes, they will most probably stumble with a common exception handling antipattern.

Exception handling antipatterns are, in most cases, the outcome of not following the previously described best practices for exception handling (details can be found on Table 1.2). And, although the community is well aware of these malpractices and knows that they are extremely widespread, there are no studies showing how exactly spread their usage is and how they affect the overall software resilience to errors.

Table 1.2 – Exception handling antipatterns.

Antipattern	Description
Log and throw	Logging and throwing results in multiple log messages for a single exception in the program. It makes log files unreadable and makes the debug task harder.
Throwing top-level exceptions	This is the same as saying that the function may have some problem without giving any hint about what that problem might be. If something happens, the client code will hardly have the chance to do anything to recover.
Throwing the unnecessary multiple exceptions	When a method declares throwing multiple exception types, it should only differentiate them if multiple treatments apply. Otherwise, exceptions should be wrapped on a single type.
Catching top-level exception	If a method declares to throw some specific exception type, it means that the calling code should handle that exception. By catching a general exception type, the client code will not be aware of what was the problem.

Antipattern	Description
Destructive wrapping	Re-throwing an exception wrapped on a different type results in the loss of the stack trace, if the original exception is not included in the wrap.
Log and return null	When a handler treats an exception by logging it and returning from the method with a null. This practice may be valid in some cases but, usually it is not. The best practice is to throw the exception and let the client code deal with it.
Silencing the exception	Also known as <i>swallow the exception</i> . It consists in providing an empty handler (or one that just returns from a method) to avoid an exception manifestation. In some cases it may be a valid behavior but, most times, it does more harm than good because it hides the reason for a possible problem.
Throw from within a finally block	An exception thrown inside a protected block will be lost forever if afterwards another one is raised inside the finally block.
Multi-line log messages	Using multiple calls to the logger inside a handler can lead to a log file where related lines may end up spaced by thousands of lines. Imagine a server app with 500 thread running in parallel and writing to the log.
Do not ignore exceptions	A method throws an exception because it is important for the client code to know that something happened. The action to take should be in conformity with the gravity of the exception.
Check for the cause	Some exception instances allow the client code to access the original exception in the base of a series of chained exceptions. But, note that there can be several exception chained simultaneously and, in this case, to access the original exception it may require more than one call to the method providing this functionality.
Throw when not implemented	If a declared method is not implemented in some class (or version of a class), be sure to throw an exception that will inform the user of that fact. Practices like just returning null will not suffice and may lead to undesired behaviors.

A fundamental part of our work was dedicated to quantifying how exception handling antipatterns *influence* software quality. We discovered that, in general, exceptions are not being correctly used as an error handling tool [Cabral2007]. This also means that if the programming community at large does not use them correctly, probably it is a symptom of a serious design flaw in the mechanism: exception constructs, as they are, are not fully appropriate for handling errors.

Exception documentation

Exception documentation also plays an important role in the quality of the code of an application. Even more if the language does not have checked exceptions, like C#, but on

which the programmer wants to correctly deal with possible abnormal situations. Checked exceptions allow the programmer to discover which exceptions may be raised by a method at compile-time but do not force the programmer to deal with them. On systems implementing the unchecked exceptions model, programmers are not forced to declare the exceptions that a method may raise. Thus, programmers are more dependent of the available documentation to ensure the handling of every possible exception. Unfortunately, exception documentation in most programs and software libraries is of poor quality or completely inexistent [Cabral2007b,Sacramento2006]. To minimize the damages, some development platforms have implemented mechanisms, on compilers and IDEs, to inform the developer of possible problems on the code. But, even these mechanisms are highly dependent of the quality of the exception handling code.

Concluding remarks

The unwillingness of software designers to deal with exceptions correctly and follow some well known best practices for exception handling will, undoubtedly, contribute to the lowering of the quality of programs and their reliability. It is obvious that something is not right with the current exception handling models: they are not adequate enough for developers.

The resilience to do proper error handling has multiple origins: the need to concentrate on the design/implementation of business code; ignorance about possible abnormal behaviors in the code; the need to speed up development; incomplete testing batteries and code coverage tests; among others. Nevertheless, these are only suggestions about what could be the problem behind exception handling poor practices. There is no sound theory about what is really keeping programmers from implementing valid recovery strategies but, as some argue, the mechanism itself can be seriously flawed. For instance, Garcia et al. [Garcia2001] have identified several design issues on the exception handling models available in modern programming languages. The authors claim that most of the existing exception handling models rely on classical design solutions, some of them too general or too complex, making harder the task of developing dependable object-oriented software. Nowadays exception models make the writing of exception handling code with quality a cumbersome task.

Some researchers have even stronger views about the application of exception handling models to object-oriented programming languages. They imply that “exception handling can contradict the conventional object oriented paradigm” [Miller1997]. The authors

identified four aspects of exception handling that are different from normal object orientation:

1. *Complete exception specification* – A handler may require extra information to be available on the exception specification than what is in the object interface;
2. *Partial states* – Object-orientation does not defines partial states but exceptions may occur in state transitions, thus giving birth to partial states;
3. *Exception conformance* – Overloaded methods have the same meaning in different situations but, exception information usually needs to be specific;
4. *Exception propagation* – Propagation can change the control flow of a program giving birth to two different execution paths: the normal execution path, and the exception handling path.

Miller et al. also describe how the exception handling model corrupts the four major elements of object-orientation: abstraction, encapsulation, modularity, and inheritance (Table 1.3). These incompatibilities will undoubtedly create difficulties to the designers of object-oriented software that are, at the same time, concerned with reliability, fault tolerance level and object-orientation.

Table 1.3 – Exception handling and the object-oriented paradigm.

OO Topic	Objective	Incompatibility with exception handling
Abstraction	Hide implementation details of an object from its users. Expose only the necessary functionality.	Many times, in order to generalize operations and make them usable in a wider range of conditions, the exception handling mechanism may require exposing more implementation details, as a part of the abstraction, to the object's users. For instance, a method on an object may declare to throw an exception type that will give more information about the method's implementation details than the user would ever know if the exception was not there.
Encapsulation	Hide the internal data and functionality of an object from outside referers/users.	When an object raises an exception, it runs the risk of exposing its internals, if the raised exception contains more information than what is permitted by the encapsulation.

OO Topic	Objective	Incompatibility with exception handling
Modularity	Ensure that the changes in one module have little effect on other modules.	Exception handling often increases the coupling between modules. Changes and evolution of the functionality of a module often require the module to expose more exceptions to the other modules than those initially planned. Thus, the other modules will have to adapt to the new exception interface members.
Inheritance	Promote code reuse and conceptual specialization.	The problem arises when a subclass's exception handling replaces rather than augments the parent's handling of exceptions.

The motivation for our work was that, quoting Garcia et al. [Garcia2001] - "We believe that an ideal object-oriented exception model is urgently needed to guide the design of effective exception handling mechanisms". In our case, we believe that the model should provide effective exception handling and do not lower the productivity of programmers; it should free the programmer from the burden of having to deal with all possible exceptions and yet keep him informed about all the potential problems that can occur in a function call; it should decrease the amount of code that the programmer effectively writes, increase code quality, speed up testing procedures/development time, and, at the same time, eliminate some common exception handling malpractices. Furthermore, we believe that exception handling mechanisms should, to an appropriate level, become transparent for the developer.

Consider the analogy with Garbage Collectors (GC). Before garbage collection became mainstream, programmers had to handle memory manually, at many locations in the source code. They had to reserve memory, free memory, and manage all memory usage related details. Automatic memory allocation and the GC freed the programmers from these tasks by automatically managing memory space as required by the running applications. This technology, besides making the job of the developer simpler, helped to avoid many memory related errors. Exception handling should work as a GC for exceptions in the sense that, without (or with minimal) programmer intervention, the mechanism should automatically execute sets of benign recovery actions for the exceptions being raised in the running code. The mechanism should also allow the running application to re-execute the problematic instructions a second time without problems or just continue its execution in a valid state.

1.2. Research Objectives

When the ideas behind this thesis first began to emerge, they were fueled by the unsettling feeling that something was not right with nowadays exception handling approaches. Current work at that time involved the study of the source code of several open source applications available on the internet. Many of these applications were servers, programming libraries, and middleware software with thousands or even millions of users all over the world. And, although we were not looking for coding errors, or coding patterns of any kind, the by-product of this task was the discovery that developers were not dedicating enough attention to the exceptions and that, in fact, the exception handling code in these programs could be much better. We believed that we could help to improve the quality of software, if we provided the right tool for exception handling to the developers. The following goals were set at the start of this investigation:

1. Identify and quantify the problems behind the general lack of efficiency on exception handling code. Assess the true influence of programmers' exception handling mal-practices on the quality of the code;
2. Investigate how the identified problems can be related to current exception handling approaches. Propose new exception handling and programming models that are more attractive to developers, thus eliminating the shortcomings of the existing models, and, at the same time, improve the resilience of programs to errors;
3. Assess the advantages and disadvantages of the new model when compared with previous approaches. For doing so, a number of prototypes and tools, either for implementing or supporting the new model, should be built.

It must be mentioned that from the start it was neither an objective to develop a new runtime environment nor a new programming language. The work should be as close as possible of the existent mainstream solutions, languages and platforms. We believe that the success of the new model is not only dependent of its novelty and efficiency but also of the easiness of integration with the most popular platforms in the market.

1.3. Contributions

The authors of this work secretly expect that their research might one day influence the development of future programming languages and runtime environments. But, for the moment, the major achievements resulting from this work are:

1. To present the most comprehensive study done on exception handling to date, providing a quantitative measure useful for guiding the development of new error handling mechanisms;
2. To provide a *novell* exception handling programming model that automates the handling of some exceptions and makes their treatment a platform issue. The proposed model uses a Software Transactional Memory (STM) [Shavit1995] approach in a way that is completely transparent for the developer;
3. To show that it is possible to define sets of benign recovery actions that can be automatically executed by the runtime platform when an exception is raised inside a running program;
4. To demonstrate that it is possible to apply the new model to existent mainstream object-oriented platforms with the advantages of: diminish the amount of exception handling code that developers have to write in their programs; decrease the development time (programmers write less code and have less code to test); increase the overall quality of the code (developers can concentrate on the writing of business code); and, at the same time, increase the software resilience to errors.

1.4. Structure of the Dissertation

The thesis is organized in six chapters:

- **Chapter 1**, this chapter, presents the motivation for the undergone investigation, initial research objectives and contributions of the thesis;
- **Chapter 2** discusses the state of the art on exception handling models and presents a brief overview of the implementations and exception handling constructs available in the most relevant programming languages;

- **Chapter 3**, provides quantitative measures on how programmers are currently using exception handling constructs in modern object-oriented programming languages. It aims to contribute to the discussion about current exception handling limitations;
- **Chapter 4** presents the Automatic Exception Handling mechanism and describes the associated programming model;
- **Chapter 5** describes how a prototype of the new exception model was actually implemented, in terms of architecture and coding, and discusses the validation process and the model's assessment results;
- **Chapters 6** concludes the thesis by summing up the major results from the research and describing venues for future work.

Current Approaches to Exception Handling

This chapter examines the current state of the art in exception handling. It addresses the origins of the concept and the first efforts, on the late seventies, for defining the architecture, language constructs, and usage semantics. The strengths and limitations of modern exception handling mechanisms are also discussed. Whenever relevant, we emphasize the features of the exception models that may have a positive or negative influence on the quality of the produced code, in terms of reliability.

2.1. Introduction

The mid-1950's saw the birth of the first exception-like language construct. The language designed by John McCarthy, *Lisp*, featured a language construct that allowed the interpreter and compiler to gracefully exit from an error when one occurred [McCarthy1965]. The function `ERRSET` permitted the controlled execution of code that might cause errors. The special form `(errset form)` evaluates the execution of `form` in a context in which errors do not terminate the program or enter the debugger. If `form` executes successfully, `ERRSET` returns a singleton list of the value. If the execution of `form` goes wrong, the `ERRSET` form quietly returns `NIL`.

Later, *MacLisp* [Eastlake1968,Moon1974] added the function `ERR` to signal errors. If `ERR` is invoked within `form`, then the argument to `ERR` is returned as the value of `form`. Unfortunately, these constructs soon began to be misused by programmers that did not use them to trap and signal errors but for execution control purposes¹. This behavior made debugging harder because unexpected errors were also trapped within `ERRSET`. In order to limit the use of `ERRSET` to error trapping, *MacLisp* designers introduced a new pair of primitives, `CATCH` and `THROW` [Eastlake1972].

This historic note is useful to help us understand that careful design of a new language construct is not enough to assure its success. In the next sections, we will present and describe several features (and their evolution) of existent exception handling mechanisms. But, despite such technical advances, it is important to keep in mind that the unintended (miss-directed) use of error trapping mechanisms, as occurred in *Lisp*, is still a problem for modern programming language designers. In many cases, designers are forced to go back and redesign the mechanism in order to comply with the users' expectations and the system correctness requirements [Steele1993]. We would have thought that the lessons learnt with *Lisp* would prevent the same from happening in modern programming languages. Unfortunately, this was not the case. Modern exception handling mechanisms are as prone, or even more, to misuse by programmers as its *Lisp* ancestor.

¹ It is interesting to note that, in many cases, modern exception handling mechanisms are currently being misused in the same way.

2.1.1. First efforts in the definition of a standard notation

Facilities for dealing with exceptional conditions, such as *variable overflow*, *end-of-file*, and *bad data*, were fairly common in the 1960's programming languages. But, it was not until the development of the *IBM PL/I* programming language [IBM1968,Radin1981] that we saw the usage of high level control flow constructs exclusively dedicated to enabling the writing of reliable and safe programs.

PL/I featured a construct, the *ON condition*, which allowed the specification of the actions to be undertaken when one abnormal condition, of a set of 23, occurred during the execution of a program (e.g., `NDFILE = 0; ON ENDFILE(SYSIN) NDFILE=1;`). The *ON* unit is not lexically associated with a statement/operation that might present an abnormal behavior. Instead, its invocation is dynamically associated with the occurrence of an exceptional condition. This construct has been proven to be difficult to use, much because there is no dedicated way to share data with the *ON* unit code. In fact, it is necessary to use global variables.

Independently of the discussed shortcomings, the *PL/I's ON condition* was useful to demonstrate that such a mechanism, or a similar one, was essential for the development of reliable software. We must recall that previously known error handling techniques had even more fallacies. For instance, *return of error codes*¹ and *status flags*² techniques have noticeable drawbacks:

1. An error is handled only when it is detected. Hence, programmers have to explicitly check/test the return values or status flags. Failing to do so will allow the program to continue its execution after an error occurrence. This can lead to the state of a program being corrupted and erroneous computation;
2. The code for testing return values or status flags has to be located throughout the program. This reduces the readability and, consequently, the maintainability of the code;

¹ This technique requires that each routine must return a value on its completion. Different values have different meanings and will indicate if an abnormal condition was encountered during the routine execution.

² The *status flag* technique might be used alone or in conjunction with the previous technique. It is based on setting the value of a shared variable (status flag) to indicate that a rare condition has occurred. The value remains until it is overwritten.

3. It is difficult to ensure that all the error cases being produced by a routine are being handled;
4. The code for testing return values and status flags is coded inline with *normal* code, thus making the removal, modification or addition of return or status values very difficult;
5. The return values technique allows the mingling of error values among the range of *good* return values of a function. Changing a value representing an exception to a *good* return value, or vice versa, can be a difficult and error prone task that will affect every piece of software using that function;
6. A function will leave the caller unaware that an error occurred if it fails to execute but still returns a valid value. This is known as the *Semipredicate* problem.

These initial efforts were followed by more complex attempts to provide the programmer with better tools to deal with abnormal situations: (a) *Subroutines* were handlers that were passed as a parameter on an operation invocation; (b) *Labels* marked the start of the handling code and were passed as parameters to operations in order to allow execution to continue on the labeled instruction after an exception detection; (c) *Object-oriented exception handlers* [Ross1967] were subroutines associated with an object that were executed when the object encountered certain conditions; (d) *Handler setup calls* [Softec1972] allowed the association of a handler with an exception being subsequently raised by some operation; (e) Hoare *otherwise* statement [Hoare1973] permitted the specification of the policy Q1 otherwise Q2 - meaning that if Q1 fails, then Q2 should be performed; there were similar techniques, such as *backtracking* [Golomb1965] and the *recursive cache* [Horning1974] methods.

In 1975, John B. Goodenough published his seminal work on exception handling [Goodenough1975] and became the first author to propose a notation for working with exceptions - the *programmed exception handling model*. His article in the *Communications of the ACM* was the first to discuss the issues associated with exception handling and the language features necessary for dealing with exceptions. And, although this work is more than three decades old, it remains up to date and many of its proposals are still found in nowadays exception handling mechanisms.

Goodenough gives a simplistic definition of what an exception is - "Of the conditions detected while attempting to perform some operation, *exception conditions* are those

brought to the attention of the operation's invoker." Although very simple, this definition is essential to understand the fundamental issues behind the *exception handling* concept. He continues by defining the *raising* of an exception as the act of bringing one of these conditions to the attention of the invoker; and classifying the invoker's response as the *handling* of the exception.

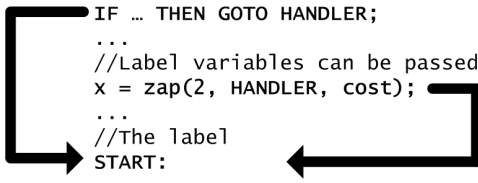
Contrary to what one might expect, the author explains that exceptions are not necessarily activated in rare occasions. They are also an elegant means of interleaving actions belonging to different levels of abstraction [Liskov1974,Dijkstra1968,Liskov1972a,Liskov1972b,Woodger1972,Dahl1972]. In essence, exceptions make operations usable in a wider variety of contexts than would otherwise be the case. Exceptions serve to generalize operations. They allow the user of an operation to extend the set of inputs for which effects are defined and its range in order to fulfill a particular purpose. Exceptions can be useful to indicate the meaning of a valid result, the conditions under which it was obtained, or to monitor the progress of an operation.

Even more important is the role that exceptions play when dealing with an operation's impending or actual failure. For instance, two failure types were identified: *range failures* and *domain failures* [Goodenough1975]. In the first case, the failure is caused by the operation being incapable (or deciding it may not ever be able) of satisfying its output assertion. In this case, the operation may need to be aborted, retried or terminated yielding partial results. Domain failure occurs when an operation's input does not respect the pre-conditions for its acceptability. This may cause the operation termination or require the modification of the input.

The notation proposed by Goodenough addressed four distinct classes of topics:

- *Association of handlers with invocations of operations* – issues related with how the association of a handler to the invocation of a given operation can be made. Handlers can be associated with blocks of code, methods, objects, classes, exception types and instances;
- *Control flow issues* – issues related with the execution flow following the execution of the handler after the occurrence of an exception, namely the applicability of the resumption or the termination model [Goodenough1975f];
- *Default exception handling* – in some cases it is useful to provide default handlers for exceptions that are not handled by the operation's invoker;

```
//Declare the label variable
DECLARE HANDLER LABEL INITIAL (ERRORS);
//Assign a label to the variable
HANDLER= START;
...
//Jump to the label yielded by the evaluation of LBL
IF ... THEN GOTO HANDLER;
...
//Label variables can be passed as arguments
x = zap(2, HANDLER, cost);
...
//The label
START:
...
```



The diagram consists of two thick black arrows. One arrow starts at the 'zap' function call in the code and points to the 'START:' label. A second arrow starts at the 'START:' label and points back to the 'GOTO HANDLER;' statement, illustrating a loop or a jump to a specific point in the code.

Figure 2.1 – Label variables usage example

- Hierarchies of operations and their exceptions – differences found between the handling of an exception by the operation’s invoker and the handling of the exception by and indirect invoker.

Goodenough proposed three different constructs for signaling exceptions: (a) SIGNAL, which permits the operation raising the exception to be either terminated or resumed; (b) NOTIFY, which forbids termination of the operation and requires resuming; and (c) ESCAPE, which forbids resuming and requires the operation termination. Furthermore, the author also proposed the use of the ENDED exception type for signaling a valid termination of an operation. Thus, allowing the execution of a handler specially created for execution after a *normal* termination.

These and other topics are, in part, what characterizes an exception handling mechanism and will be addressed in the following sections of this chapter where a discussion about all the attributes of modern exception handling models will take place.

2.2. Handling models: features and propagation

The literature on exception handling has already provided a thorough description of the existent exception handling models and their attributes. This section discusses the relevant literature in order to give a general, but clear, perspective of the capabilities of nowadays exception handling models.

```

try
{
    // Code executed
    ...
    ★ Exception raised
    // Code not executed
    ...
}
catch (...)
{
    // Code for handling the exception
    // and replace the code with problems
    ...
}
// Code after the protected region and the handler
...

```

Figure 2.2 – Java code exemplifying the termination model

2.2.1. Handling models

Exception handling models are differentiated by their control flow policies. An exception handling mechanism can implement more than one exception handling model, their usage is not mutually exclusive. Yemeni et al. [Yemini1985] identified the following models:

- Nonlocal transfer* – few programming languages support nonlocal transfer. PL/I is one of those languages, it uses *label variables* as arguments for `goto` statements in order to redirect the control flow. The label variables contain both a *point of transfer* and a pointer to an activation record on the stack containing the transfer point. An exception handling mechanism that uses the nonlocal transfer model can be constructed by labeling code, to identify handlers, and perform branches to those labels for terminating operations - Figure 2.1. However, this model suffers from a well known structured programming problem: it allows branching to almost anywhere, making the code difficult to reason about, less maintainable and error prone.
- Termination model* – this is the most commonly used model. When an exception is raised inside a protected block of code, control flow is transferred to a handler and the intervening blocks are terminated; after the completion of the handler, control flow continues as if the operation in the protected block terminated without showing any abnormal symptoms - Figure 2.2. The handler is, in this model, an alternative set of operations that are executed after the problematic ones in the protected region.

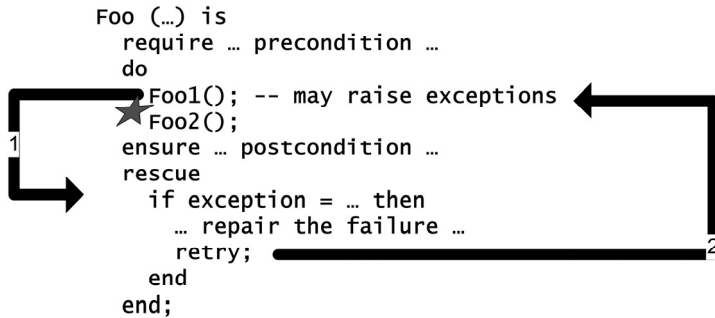


Figure 2.3 – Retry model exemplified with Eiffel notation

- Retrying model* – this model extends the previous model. It allows the failing operation to be re-executed after the execution of the handler for the raised exception. Usually the execution is retried from the beginning of the protected region of code where the exception was raised or from the beginning of the one associated with the handler used to treat the exception - Figure 2.3. The re-execution of code blocks can have unpleasant side effects that the programmer must be aware and that have to be dealt with in the most appropriate way: counters not reset; invocation of non-idempotent operations; existence of several handlers for the same reentry point; etc. Other authors [IBM1981] have shown that it is possible to mimic the retry model by using a loop and the termination model. Because the later alternative is more readable, all looping is the result of explicit loop instruction and not from a hidden language feature; its use has been considered preferable when compared with the model itself.
- Resumption model* – the control flow in this model is in many ways similar to a normal routine call: when an exception is raised, control flow is transferred from the raise point to a handler, to treat the problem at hands, and then back to the raise point - Figure 2.4. In fact, the main difference between a normal routine call and a resuming call is that in the second one the handler is located dynamically. The main argument against the resumption model is its complexity. The Goodenough model and the Mesa model [Mitchel1979,Yemini1985] (based on the first) are good examples of that complexity. Implementation difficulties and complexity apart, the remaining problem with this model is *recursive resumption* [Liskov1979,Stroustrup1994]. Recursive resumption occurs, for instance, when a handler for a resuming exception resumes the same event.

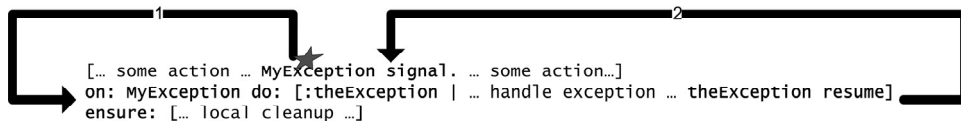


Figure 2.4 – Resumption model exemplified with Smalltalk notation

2.2.2. Features

Control flow issues are fundamental when distinguishing exception handling models and mechanisms. However, there many other features that can make a difference when dealing with exceptions. Burh et al. identified a set of the most fundamental features available in modern exception handling mechanisms [Burh2000]. We will discuss these features on the following sections.

Derived Exceptions

An exception type can derive from another exception type, much like a class can derive from another class in the object-oriented paradigm. This allows the developer to organize his exceptions in a way similar to a class hierarchy. Furthermore, by doing so the developer is able to handle an exception at different degrees of specialization along the hierarchy. This feature allows a more flexible programming style.

Multiple derivation [Koenig1990] is a feature that is often rejected in object-oriented languages due to the complexity and problems that it introduces. Multiple derivation for derived exception types is also rejected because it would create significant difficulties for the mechanism semantics. Consider the example on Listing 2.1. In this example, a new exception class, `InputException`, is declared as being derived from the classes `IOException` and `EOFException`. The difficulties arise when an exception instance of `InputException` is raised inside a guarded block that has two handlers, one for `IOException` exceptions and another one for `EOFException`. This situation causes ambiguity: without the specific knowledge of the exception hierarchy it is not possible to say for sure which of the handlers will be activated. Moreover, in the event that one of the handlers (or both) is executed, none of the parent classes might be equipped for dealing with the derived exception, but, are forced to do so.

```
class InputException extends IOException,EOFException {...}
...
try
{
    // operation raises an InputException
}
catch (IOException ...)
{
    // will this handler be activated?
    // if so, does IOException have
    // enough information/functionality
    // to deal with the descendent exception
}
catch (EOFException ...)
{
    // Or this one?
    // would this be able to handle it?
}
...
```

Listing 2.1 - Multiple derivation for derived exceptions

Exception Parameters

Raising an exception would be almost irrelevant if the exception handling mechanism did not allow the passing of information about the problem to the exception handler. This could be accomplished by using shared variables but, in environments with concurrency concerns, the usage of locks or other access control mechanisms would be mandatory.

Exception parameters allow the source code to pass information to the handler without any side-effects or locking requirements. There are, of course, mechanisms without exception parameters and mechanisms that, on the other hand, allow passing one or more parameters.

In concurrent environments, where the *faulting execution*¹ is different from the *source execution*², the access to the exception parameters must be properly synchronized. Burh et al. [Burh2000] defend that synchronization should not be a concern of the exception mechanism - “leaving the synchronization to the programmer simplifies the exception handling mechanism interface and hardly loses any capabilities” - the programmer can use monitors, futures, conditional variables, and other facilities for synchronization.

¹ The execution that changes its control flow due to a raised exception. Control flow is routed to the handler.

² The execution raising the exception.

```

class stack
{
  exception overflow;
  boolean empty;
  ...
}

...
stack s1, s2;
try
{
  ...
  s1.push(...);           //may raise overflow
  s2.push(...);           //may raise overflow
  ...
}
catch ( stack::overflow ) ...           //option 1
catch ( s1.overflow ) ...               //option 2
catch ( s2.overflow ) when (s2.empty) ... //option 3
catch ( s2.overflow ) when (!s2.empty) ... //option 4
...

```

Listing 2.2 – Bound exceptions and conditional handling

Derived exception parameters deserve special attention to prevent handlers from accessing un-initialized parameters on the exception object.

Bound exceptions and conditional handling

When an exception is raised inside a guarded block and a handler is activated, the handler should know which object was the origin of the exception. In Ada, this question has a simple answer: an exception declared in a generic package creates a new instance for each package instantiation. Thus, a different handler can be activated for each different package. On the other hand, in other systems, it may be necessary to pass additional information from the source code that generated the exception to the handler.

Consider the example in Listing 2.2. In some models it is possible to declare an exception in a class (e.g., setting up the exception `overflow` on the `stack` class). This exception can then be interpreted by the exception handling mechanism as being associated with the class or with each object instantiated from it. In the source code example, the handler marked as *option 1* acts like there is only one `overflow` exception for all `stack` objects created. Hence, this handler is activated regardless of which `stack` object raises the exception. The handler marked as *option 2*, on the other hand, is activated only if the object raising the `overflow` exception is `s1`. This is what is called a *bound exception*; the exception `overflow` is bound to a particular class instance.

Bound exceptions can be mimicked in systems that do not support them. Unfortunately, the cost in terms of code growth can be too high. For instance, consider that (1) a new exception instance would have to be created for each `stack` object, (2) that the `stack` object reference, of the object raising the exception, can be passed to the exception and, subsequently, to the handler through the exception parameters, (3) and that the programmer would have to check if the object referenced by the exception is the one that he or she wants to handle (comparing references) and, if it is not, he or she is obliged to re-raise the exception in order to let the search for the appropriate handler continue. This last coding convention is also considered to be unreliable. Mimicking the bound exceptions feature is unfeasible for derived exceptions, especially in the cases that the exception has to be re-raised, after entering in an inappropriate handler (set for a parent exception), and ignores the right handler if it is also associated with the same guarded block.

The concept of bound exceptions can be extended with *conditional handling* [Mok1997]. *Options 2 and 3* of Listing 2.2 are a good examples of how conditional handling works. By checking the value of the variable `empty`, the system can decide which handler is activated. Furthermore, conditional handling can be used to mimic bound events just by checking if the object parameter is equal to the desired object. While, the literature has already given us proofs of the usefulness of bound exceptions [Burh1992], we have none on conditional handling.

The .NET framework, for instance, provides four different kinds of handlers:

- **Fault handlers** are called whenever an exception occurs. After their execution the exception keeps propagating up the stack;
- **Type-filtered handlers** handle exceptions of a specified type or a subtype of that type, and are executed when such an exception is thrown;
- **User-filtered handlers** decide whether to handle an exception or not based on custom logic; if the test passes the handler will cope with the exception, otherwise the exception survives;
- **Finally handlers** are executed under any circumstance, regardless whether an exception occurs or not and can be used to close critical resources such as files or handles.

User-filtered handlers are the .NET implementation/construct used for conditional handling. Unfortunately, user-filtered handlers are not available either in C# or in Visual Basic, two of the core programming languages of the .NET platform.

Exception list

The exception list is an extremely useful feature that allows a method to declare which exceptions it might raise. For doing so, the programmer has to include in the method's signature a list of exception types. In most programming languages, by doing so, the programmer is explicitly saying that the method in question can only propagate in the call stack the exceptions being declared. Any other attempt of throwing or propagating an undeclared exception would result in a compile error. This feature gives the developer the privilege of always being aware of the potential problems that his code might face when in execution and avoids the existence of unattended exceptions. An exception list is also very useful for the development of static code analysis tools because it allows the observation of potential exception propagation paths.

Some developers think of this feature as a restriction for their coding styles. They claim that is not always useful to handle all the declared exceptions and re-declaring the not-handled ones in the method signatures is, in some sense, a unnecessary and *ugly* coding practice. Furthermore, this technique can raise some difficulties for object-oriented software designers. For instance, when overriding an inherited method and the new implementation does not throws the same exceptions as the parent method, it is impossible to add or remove items on the inherited exception list. It would be possible to add the necessary exceptions (to cover all possible exceptions in all existing implementations) to the declaration on the top of the hierarchy but, this practice would make the program less reliable because the signature would cover a large range of exceptions.

Propagation mechanisms

There are two kinds of propagation: *throwing* and *resuming*. The first is associated with the termination model while the other one corresponds to the resumption model. Exception handling mechanisms are free to implement only one or both of them.

In the *throwing propagation*, the execution flow of the program never returns to the point where an exception is raised immediately after the handler execution. Furthermore, the propagation requires that every block in the stack, between the raise and the handler, must be destroyed. This is known as *stack unwinding*.

In the *resuming propagation*, the execution flow returns to the point where the exception was raised after the execution of the handler, thus, there is no stack unwinding. However, the handling code is free to decide not to resume the execution and perform the *unwinding* of the stack. This ability to explicitly request the stack unwind must be supported by the programming language (a special statement must be available). This feature may lead to unsafe resumptions given that the source code (where the exception is raised) loses the ability to order the unwinding of the stack.

Liskov and Snyder [Liskov1979] have discussed the suitability of each approach in terms of software reliability. Their findings do not come as a surprise. For instance, in terms of control flow (and in the absence of recursion) it is safe to assume that the caller of a function is dependent of the function being called but, the called function is not dependent of its caller. Nonetheless, with *resuming propagation* the later can happen, and both the caller and the callee can be mutually dependent. For instance, not only the caller of a function is dependent of the invoked function but also the callee becomes dependent of the caller when an exception is raised (and the handler is located on the caller). This characteristic influences the way applications are designed. It is necessary to include extra information on the design that will identify, not only every possible termination state (when non recoverable exceptions occur), but also all possible behaviors that can be expected from the handlers when exceptions are signaled.

Resumption requires additional language support. The “normal case” in a resuming environment is having exceptions that are resumable but this is not always possible. Every system has to cope with situations where the signaler of an exception may not be resumed, must be resumed, or where resumption is optional. Language designers have to outfit their languages with the necessary tools to support these three possibilities and differentiate the signals used to communicate them. Furthermore, in the event that an exception is not resumable, the signaler must have the means to restore the global variables to a valid state and perform all the necessary clean up actions before its activation is terminated. This kind of system also requires a *termination-like* functionality in order to handle its own exceptions in the event that the caller does not. Thus, the termination model has simpler linguistics and does not require multiple kinds of signals. On the other hand, it is also true that the resumption model will provide a more expressive (and more intuitive) way to deal with exceptions in specific scenarios such as the following:

- “when the exception is signaled, the signaler is in the middle of a computation that can be completed by performing additional computation upon receipt of a

value from the handler. Resumption permits completion of the computation in this situation without redoing work already performed” [Liskov1979]

In [Goodenough1975, Levin1977] the authors present strong cases in defense of the resumption model. Nonetheless, their examples are simpler deviations of the scenario just described. It is arguable that the solutions using resumption feel more natural (or not) than those possible without resumption. But, even if resumption can relieve the programmers’ task in these cases, it is necessary to measure how common these scenarios really are to know if the tradeoff between the added extra complexity and the more natural code in such specific conditions is justified.

When developing software for a platform using an exception handling mechanisms that implements both propagation mechanisms, the programmer must have in mind that a new class of problems will arise. Consider the following example: in a program it is possible to have a throw statement overriding a resume statement. Nonetheless, it is not possible to have a resume statement overriding a throw statement because the stack is already unwind after the throw.

Until now, it has been assumed that the handling model, termination or resumption, matches the propagation mechanism, throwing and resuming respectively. But, this is not necessarily always the case. For instance, a handler with resuming semantics is not able to handle a thrown exception because a terminated operation cannot be resumed. On the other hand, the best option for a handler under the termination model to handle a resumable exception is to unwind the stack and follow the termination semantics.

Handler search

One fundamental issue, associated with propagation mechanisms, is the selection of a suitable handler for an exception occurrence. Most systems adopt what is known as *dynamic propagation*. The alternative method is named *static propagation*. In dynamic propagation, the call stack is searched to find an appropriate handler. The second one, static propagation, performs the search on the lexical hierarchy of the program’s code.

Dynamic propagation is often the best guarantee that an exception will be handled near to the point where it was first raised, and by the handler that is closest to the block where the propagation started. It is even feasible that the block on the top of the stack will be elected to handle the event if it has an appropriate handler.

```
class A
{
    virtual int g() {}
    int f()
    {
        ...
        g();
        ...
    }
};

class B: public A
{
    int g() raises(E) {raise E;}
    int h()
    {
        try {
            ...
            f();
            ...
        }
        catch (E) { ... }
    }
};
```

Listing 2.3 – Dynamic propagation through an invisible scope.

Handling an exception high in the call stack allows the handler to perform more specific actions, whether, if the exception is handled down in the call stack a more general (accordingly to the higher abstraction level) action would be in place. Most times, it is easier to handle an exception in a specific context than in a more general context. Dynamic propagation also minimizes the extension of stack unwinding.

However, dynamic propagation can cause an exception to be propagated through a block in a different lexical scope. For instance, in C++ it would be impossible to provide an exception list within the declaration of a template routine. The problem is that there can be many implementations for the same routine and the exceptions thrown by each one of them can be different. Consider the template routine `template<class T> void sort(T items[])`. This routine uses the operator routine `<` to compare pairs of items on the list (`bool operator<(const T &a, const T &b)`) and sort them. The implementation of the routine `operator <` is dependent of the objects being compared thus it is impossible to know which exceptions can be raised in advance. Another example, adapted from [Burh2000], is presented on Listing 2.3, it shows that while method `B::h` is equipped to deal with exception `E`, method `A::f` is not even aware of the exception. Since `B::h` invokes `A::f` and `A::f` invokes `B::g`, `A::f` will propagate `E` without even knowing the exception. This is an undesirable behavior [Motet1996]. Some designers suggest that “an exception


```

try
{
  resume R;           // T(H(R)) → try block with handler for R
}
catch ( R ) resume R; // H(R) → handler for R

```

Listing 2.4 - Recursive resuming example

should never be propagated into a scope where it is invisible, or if allowed, the exception should lose its identity and be converted into a general failure exception.” [Burh2000]

Dynamic propagation allows handlers to be selected dynamically, thus, the handler chosen to deal with an exception cannot be identified through static analysis. But, at the same time, it is this functionality that allows software libraries designers to develop an API without providing handlers for the exceptions raised in their software.

Recursive resuming is a problem that can arise in systems implementing dynamic and resuming propagation. Mostly because, in these systems, due to the dynamic choice of handlers, it is difficult to discover, both at runtime and compile time, the existence of problem.

The simplest example that can illustrate the recursive resuming problem is presented in Listing 2.4. In this example, the try block resumes R and consequently the handler $H(R)$ is activated and also resumes R . Considering that the blocks in the stack are organized as follows: *bottom of the stack* $\rightarrow \dots \rightarrow T(H(R)) \rightarrow H(R)$; the handler for the latter resuming R is located just above itself in the stack, it is $T(H(R))$. Thus, $H(R)$ is called again and continues to be invoked until the stack overflows.

There have been attempts to prevent recursive resuming from occurring, since it is the only serious problem attributed of resuming propagation. This was first done by Mesa [Mitchel1979] designers and latter by Burh et al. [Burh2000].

The Mesa’s approach consists in marking every handler activated for an event as being *unhandled* and not re-usable. These marked handlers cannot not elected twice for the same block. Besides its conceptual simplicity, this approach as been classified confusing when used in practice and the source of semantic negative attributes: language designers are concerned that, at certain moments, it is difficult to know if an exception generated inside a handler will be handled by blocks bellow or above it in the stack. To use resuming programmers have to have knowledge about the internals of the libraries they are using,

because some exceptions might be handled by higher stack blocks (abstraction violation.) Exceptions are not only being communicated from the *callee* to the *caller*, but also in the inverse direction.

Burh's approach introduced two new concepts, *consequent events* and *consequential propagation*:

- *Consequent event* – the raising of an exception constitutes an event. Sometimes, a handler deals with events by raising a new exception (new event). This second event is considered a *consequent* of the first one.
- *Consequential propagation* – is a different propagation mechanism that eliminates part of the semantic confusion associated with the approach in Mesa. Consequential propagation goes through the stack in the normal way but marks all the inspected handlers as ineligible (even the chosen handler). This way, any consequent event will see the marks and will be unable to use any of the marked handlers. The marks are cleared only after the event has been handled. The propagation is simplified by the fact that non-resumable exceptions cause the stack to unwind, thus eliminating the need for marking.

Static propagation was proposed by Knudsen [Knudsen1984,Knudsen1987] and promised to solve the dynamic propagation problems. This approach was based on the Tennent's *sequel* construct [Tennent1980]. A sequel is similar to a routine in many aspects but possesses a fundamental difference: when a sequel ends, execution continues at the end of the block where the sequel was declared and not after the sequel call.

Using sequels to handle exceptions is a guarantee that propagation is done along the lexical hierarchy (because of the static name binding), hence, for each exception occurrence the respective handler is known at compile-time. Unfortunately, static propagation is only able to solve dynamic propagation issues for monolithic applications: sequels cannot be referenced from code separately compiled. This difficulty can be overcome by passing the sequel as parameter when invoking the pre-compiled code. Thus making the handler selection *dynamic* (only the propagation search is eliminated). Furthermore, declarations and calls will need potentially more arguments putting additional execution cost in every call. This propagation mechanism has not succeeded on replacing dynamic propagation. The reason, besides the described shortcomings, is that it is possible to mimic the model's

```

C++
int foo;      // outer
try
{
  int foo;    // inner
} catch (...)
{
  foo = ...   // outer
}

Ada
VAR foo: INTEGER;           -- outer
BEGIN
  VAR foo: INTEGER;         -- inner
  EXCEPTION WHEN Others →
    foo := ...              -- inner
END;

```

Listing 2.5 - Handler's static context in C++ and Ada

syntax and semantics using some advanced language features, such as generics and overloading, and other exception handling features.

Handler's context

The static context of handlers can vary from one programming language to another. For instance, in C++ and Ada the scope of the referenced variables is very different: while Ada's handlers are nested inside the guarded block, C++ handlers execute in a scope outside of their guarded block. Listing 2.5 illustrates the differences in handlers' context between the two languages.

Handler and exception partitioning

An exception handling mechanism can implement both termination and resumption models. As a consequence, programming language designers proposed that it should be possible to declare at compile-time which exceptions and which handlers act in accordance with the termination model and which use the resumption policy. This feature is known as handler and exception partitioning.

Handlers can be declared at compile-time as being either *resuming* or *terminating* [Gehani1992,Burh1992,Madsen1993]. For doing so, the general `catch` statement is replaced by two new clauses: `resume` and `terminate`. An exception thrown inside the guarded block is handled by the terminating handler and a resuming exception by the resuming handler.

If the programmer prefers to delay the choice of the type of handler until run-time, a different way of achieving the same principle is the usage of a state flag declared on the global application context. If the flag is set for *resumable*, the handler should provide for a resuming exception, otherwise, a terminating handler is chosen.

In some circumstances, an exception can be handled by the wrong type of handler. The partitioning of exceptions can help avoiding these situations. Exceptions can be *throw-only*, *resume-only*, or have a *dual* nature (the default) [Goodenough1975]. Throw-only exceptions can only be handled by terminating handlers, resume-only exceptions can only be handled by resuming handlers, and dual exceptions can either be thrown or resumed.

The separation in the nature of exceptions is potentially beneficial because it increases the expressive power of exceptions. For instance, the Unix SIGTERM and SIGBUS signals always lead to the termination of an operation and hence, should be declared throw-only.

Some problems arise regarding the programmability of a hierarchy of partitioning exceptions. For instance, consider the case where a parent exception is throw-only and the child exception is resume-only. If the derived exception is thrown but the parent exception is caught, the stack is unwind and the resume point of the child exception is invalidated. The inverse, where the parent exception is resume-only and the child throw-only, is also problematic. If the throw-only exception is raised but the resume-only is caught, the event could be resumed but the termination at the raise point is invalidated.

Handler selection

The selection of a handler during propagation obeys the *three orthogonal criteria*: *agreement*, *closeness*, and *specificity* [Burh2000]. The first two criterions are straightforward but, the last one can be difficult to evaluate.

- *Agreement* – This criterion is applied in systems with more than one propagation mechanism and assures that the selected handler matches the propagation mechanism;
- *Closeness* – This criterion decides which handler is selected due to its proximity with the raise point. A handler is considered to be closer than the rest if, accordingly to the propagation mechanism, it is located prior to others on the stack and is able to deal with the exception;

- *Specificity* – In the event that more than one handler is considered eligible, after consideration of the previous two criterions, the *Specificity* criterion is used to decide which is more specific. For instance, if both handle the same exception, the one using conditional handling is considered more specific; the handler for a derived exception of the exception being handled by a second handler is considered more specific.

Sometimes, it is difficult to declare a handler as being more specific than another: a handler for a particular exception is considered as specific as a handler for a parent exception of the former but using conditional handling.

To avoid potential conflicts on handler selection, language designers have to set priorities for each one of the criterions. Normally, agreement has the highest priority followed by closeness. Specificity comes last. On cases where two handlers on a specific handler clause are equally specific, the system opts by the activation of the one that appears first in the clause declaration.

Catch-any and Re-raise

Almost all known exception handling mechanism allow its users to specifically catch some type of exception and to raise some specific exception. But, simple mechanisms are sometimes the most useful ones. In some situations, the desired behavior is to have the ability to catch any type of exception and re-throwing it afterwards, without losing any information about the original exception. For instance, when an exception is not handled on the raising block, it is still possible to perform some kind of finalization/clean-up tasks before re-raising it.

This feature is also useful on the resumption model. It allows the gathering of extra-information before resuming normal execution flow.

Checked Vs Unchecked Exceptions

The *checked exceptions model* is clearly influenced by the exceptions list feature. In this model, the exceptions weight in the functionality of a method is considered so significant that they must be explicitly declared, being the programmer forced to handle them. On the other hand, the *unchecked exceptions model* allows developers to ignore all the exceptions that a method throws. The discussion surrounding both models has been going on for almost a decade but, no consensus has come from it yet. Given these approaches, at

present time, this discussion also means Java's vs. .NET's way of dealing with exceptions. .NET uses exclusively unchecked exceptions and Java uses both kinds, unchecked exceptions for dealing with runtime abnormal situations and checked exceptions for the rest.

In the checked exception model, programmers have to declare the exceptions that a given method, `m1`, throws (e.g., `void m1() throws IOException {...}`). There are also special instructions used to explicitly raise exceptions in the method's body – such as the Java or the C# [ISO23270:2006] `throw` instruction (e.g., `throw new Exception("");`). In most platforms, by using these constructs the exception information is naturally bound to `m1` and becomes connected to that method (it can even be accessed through reflection in reflection-enabled systems). This also has the effect of forcing programmers of another method, `m2`, which calls `m1`, to either setup a `try-catch-finally` block to handle `m1`'s possibly thrown exceptions, or declare `m2` as thrower of those exceptions, using the same process as for `m1`. For instance, a Java compiler will refuse to compile a program in which a programmer does not use one of these possibilities for all exceptions that methods called by that program are declared to throw.

On systems implementing the unchecked exceptions model, programmers can, if they wish to, declare a method as thrower of an exception, but, the relation between a method and the exceptions it can throw is weaker because the programmer is not forced to do so. Plus, another programmer, reflectively accessing a method entity, has no possibility of discovering which exceptions it may throw if the developer of that method opted for not declaring the potential exceptions¹. Nevertheless, a `throw` instruction can still exist, which means programmers can use it to raise exceptions.

The checked vs. unchecked exceptions discussion has had numerous interesting episodes. Ryder and Soffa [Ryder2003] present an historical overview of some of the older ones, stating that “there is a symbiotic relationship between software engineering research and the design of exception handling in programming languages”. What is interesting to notice is that Ryder and Soffa end by saying that “strong typing in programming languages, desirable in new language designs, was a direct answer to concerns about software reliability and correctness” which agrees with Goodenough's advocating of “compile-time checking of the completeness of exception handling”. This means that for at least the last 30 years, checked exceptions have been regarded as good for reliability. But the modern

¹ In some cases the information about exceptions might be available on the software documentation.

try-catch construction only appeared a little over 15 years ago in C++ [Koenig1993], and Java, the most popular language to use checked exceptions, in 1996. This means that hands-on experience with this topic is still recent.

Robillard and Murphy [Robillard2000] are critics of the checked exceptions approach. Using a practical example, they conclude that “although checked exceptions have many benefits, they can be expensive to implement”. This is due to the fact that checked exceptions force programmers to alter every method in the call chain, connecting an exception thrower to an exception handler, whenever the group of types of exceptions possibly thrown is modified. In the presence of large method call propagation graphs, this is impractical. It can be argued that checked exceptions clutter the object interfaces and induce complicated catch blocks. On the other hand, checked exceptions could be considered as a required language feature for ensuring reliability in applications. In the absence of documentation or in the presence of bad documentation, not checking for exceptions could mean not documenting exceptions.

The unchecked exceptions approach, defended by Microsoft, seems to imply that the programmers can be trusted to document exceptions. Sun believes that this is unreasonable and that a mechanism to enforce reliability is in order. Either way, the special documentation tags that both companies introduced tell us that they also agree on the importance of good exception documentation.

Cheng et al. [Cheng2005] shows that checked and unchecked exceptions can be consistently used: “through the use of an architectural model, an application can benefit from a separation of exceptions in terms of recoverability beyond distinguishing checked and unchecked exceptions.” The architectural models presented in [Cheng2005] help to evaluate and balance conflicting quality requirements such as modifiability, readability, and reliability. The models are useful to guide developers in using checked and unchecked exceptions.

Concurrent exception handling

Exception handling in concurrent systems differs significantly from sequential exception handling. Moreover, we believe that exception handling mechanisms should rely on the way the system is structured and be an integral part of system design. This raises some difficulties in terms of concurrent execution of handlers, exception signaling and communications between handlers. And, although the development of exception handling

models for sequential object-oriented systems has a long history, the same is not true for concurrent object-oriented systems. Research in this area is still very active and most concurrent system nowadays still use sequential exception handling.

It is difficult to design, analyze, modify and, sometimes, understand concurrent object-oriented systems. Thus, in many situations it is not possible to guarantee that erroneous information is always contained inside an object. In such systems, and in the presence of an abnormal situation, we will most probably have to deal with several interconnected objects simultaneously. In *Client-Server* architectures is not uncommon to observe server errors affecting several client objects. Dealing with the error only on one side (the client or the server) is unfeasible in most cases.

Concurrent systems can be designed to work independently (disjoint), competing or cooperating. Competitive systems are composed of two or mores individual components developed independently one from the other; which run not aware of each other, but use the same passive components (competitive concurrent activities). Cooperative systems are designed to work together to accomplish a joint goal. The components of these systems can communicate among themselves and share results or functionality.

Concurrent systems, when organized in small execution units, are easier to build, understand, and able to deal with complexity in a scalable way [Best1996,KurkiSuonio1997]. These units can encapsulate objects and method calls. Hence, assuring that no information crosses the units' borders. The nesting of atomic units provides for a scalable growth in complexity and, at the same time, for the confinement of error information inside the unit boundaries, facilitating reliability procedures [Romanovsky1999].

Concurrency introduces new challenges for systems development and concurrent exception handling is also a main concern when designing such systems. More than confining developers to the usage of sequential exception handling techniques, language designers face the challenge of integrating exception handling into a new complex environment in a way that respects the structure of programs, their organization and goals. Atomic units [Romanovsky2001] provide the perfect context for implementing concurrent exception handling mechanisms. They allow the definition of dedicated handling policies for the abnormal events occurring on the actions performed inside the unit. They avoid the linking of the exception effects outside the boundaries of a unit and provide an elegant way of communicating to the outside the failure of a unit's execution.

In terms of concurrency, transactions have been the *battle horse* for many years. They provide for the atomicity, consistency, isolation and durability (ACID) [Gray1993] properties required on systems that access resources concurrently (competitive systems). But, even if the development language has exception handling capabilities, it is usual for these to be completely separated from the transactional structure of the system. For instance, in *Multithreaded Transactional* (MTT) systems [Kienzle2001b], such as the one in CORBA [OMG1996], developers have an extremely powerful transactional service but they still have to use the sequential exception handling mechanisms available in the selected programming language (e.g., C++ or Java). By doing so, developers have to deal with some anti-paradigmatic problems. For instance, the raised exceptions are certain to cross transaction boundaries if not handled inside the transaction. Each participant in a transaction deals with its own exceptions separately and the exception context does not match the transaction context. Furthermore, the transactional environment is no longer viable (valid or consistent) if an exception crosses the transaction boundaries.

Platform and language designers have already attempted to introduce exception handling functionality inside MTT models by giving transactions the ability to explicitly re-raise an exception, abort a transaction or deal with the exception and continue execution (e.g., EJB [Sun2006]). Nevertheless, even with such improvements, transactional systems are still not able to cope with complex exception handling procedures that require inter-transaction communication. The simplest solution to incorporate exception handling into concurrent competitive systems is to separate the exceptions that are raised and handled inside a transaction from those declared on the transaction interface (external exceptions). Some systems implement this model by declaring methods as atomic transactions (e.g., Argus [Liskov1988]), thus enabling the definition of external exceptions on the method's interface, and forcing threads to synchronize after each commit or abort. Others simply abort the transaction if an exception is not handled inside the transaction boundaries (e.g., Vinari/ML [Haines1994] and Drago [Jimenez2000]). When the participants of a transaction are unable to deal with an abnormal event locally and the transaction is aborted, all the calling threads (the invokers of the transaction participants) are informed of the exception occurrence [Kienzle2001a, Issarny1993]. Hence, the calling threads will deal with the signaled external exceptions independently.

In the atomic actions scheme [Campbell1986] several participants join an action and cooperate to achieve a joint goal. This is the essence of cooperative systems: atomic action participants can share data and work in order to complete their objective with the

guarantee that no information will cross the action boundaries. Furthermore, atomic actions can be divided into smaller and nested actions, thus providing for a higher degree of scalability. The major difference that atomic actions introduce when compared with the simpler transactional execution units is the fact that if an error is detected inside an action, all participants take part in a cooperative recovery instead of each one trying to overcome the problem independently.

Each atomic action has a set of internal and external exceptions. The external exceptions of a nested action are seen as internal exceptions on the containing action. Errors occurring inside an action can affect all the participants in that action. Thus, when it is necessary to deal with an exception in one single participant, all the remaining action members are called to intervene. Each participant has its own set of exception handlers (for the action's internal exceptions). These handlers, as happens in *normal* execution, cooperate when dealing with abnormal events in order to return the system to a consistent state. Conversations [Campbell1986,Randell1995] are a special instance of the atomic actions scheme. The conversation scheme represents a natural evolution, necessary to extend the concept of recovery blocks¹ to concurrent execution environments. It provides the means to allow backward error recovery in concurrent systems while avoiding the *domino effect*².

Coordinated Atomic action (CA action) [Xu1995] is a mechanism for structuring fault-tolerant concurrent systems that unifies the notions of forward and backward error recovery. Concurrent software systems, most times, involve both competitive and a cooperative components. CA actions provide a means for dealing with abnormal events in systems with this dual nature by enclosing and coordinating interactions among threads. CA actions combine and extend the previously discussed concepts of atomic actions and atomic transactions³.

¹ Section 2.4 gives more information on backward error recovery mechanisms and recovery blocks.

² Backward error recovery mechanisms allow a running application to revert to a previously known valid state (checkpoint) in the occurrence of an error. "However, if recovery and communication operations are not performed in a coordinated fashion, then the rollback of a process can result in a cascade of rollbacks that could push all the processes back to their starting points – the domino effect. This causes the loss of the entire computation performed prior to the detection of the error." [Randell1995]

³ Atomic actions allow the system to recover cooperatively while transactions are used to maintain the consistency of shared resources.

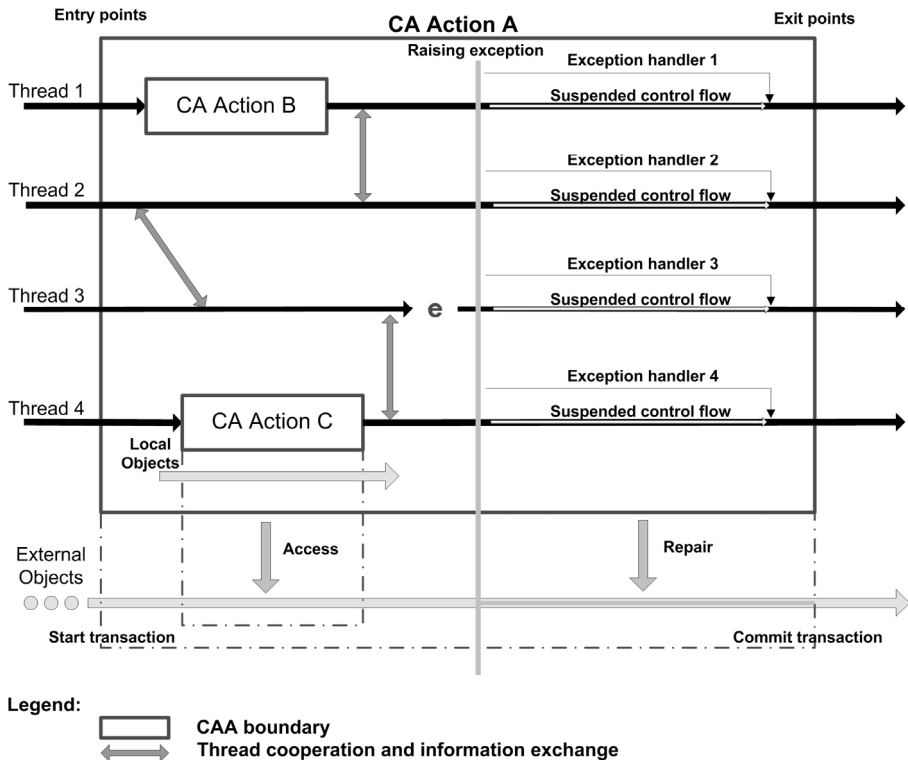


Figure 2.5 - Coordinated Atomic Actions scheme overview

Figure 2.5 illustrates the fundamental concepts involved in the CA actions scheme. The execution of a CA action looks like an atomic transaction for the outside. CA actions can be nested inside other CA actions. Concurrently nested CA actions behave like nested transactions with respect to external atomic objects involved in transactions with their parent action. Inside of a transaction, participants cooperate and interact through local objects. In the presence of an exception each participant is forced to handle it independently of the fact of which participant first observed the abnormal occurrence. Internal exceptions raised concurrently are resolved using a resolution graph. The exception graph approach is used in order to find the exception that covers all the exceptions raised concurrently, the exception that is the root of the smallest sub-tree containing all the raised exceptions.

Not all CA actions conclude their execution successfully, sometimes it is not possible to achieve the goal set for an action and execution ends abnormally. In such cases, the CA action interfaces can contain one or more abort exceptions (when signaled the CA action is aborted and local objects destroyed), a predefined failure exception and a number of

exceptions corresponding to partial (committed and consistent) results which the action can provide. This is useful to inform the containing CA action of the impossibility of producing the required results. The state of all transactional objects is aborted when an interface exception, corresponding to a partial result, is signaled.

Even the CA action scheme has its limitations. External exceptions are explicitly signaled from a CA action participant thus, in some cases, it might not be possible to detect abnormal conditions outside the participants. Furthermore, concurrently signaled exceptions are expected to be related in some way so that the exception resolution mechanism can pull off a meaningful result. But, the main problem with concurrent exception handling is determining which is the correct handler to invoke (it may be different for each participant) – “Relying on handler communication to ensure the correct handlers are invoked may be a highly complex task”- The authors of the Guardian model [Miller2002] have addressed this issue. The Guardian model, contrary to what happens in the prior model, does not raise the same exception in all action participants to notify the occurrence of an abnormal event. By raising in each process a possibly different exception and specifying the context in which it should be handled by the process, the Guardian model guides each process to a correct exception handler, thus *orchestrating* the recovery action. Miller et al. demonstrate the concepts behind their model by providing a simple example: “... say there is a pipeline of three processes A, B, and C. Should B fail, the guardian would signal to A an exception that its downstream neighbor has failed, and to C an exception that its upstream neighbor has failed. With a guardian, participants are freed from the burden of maintaining any configuration information and relating it to a process failure to determine the semantically correct recovery action. No transaction-like structure is needed for the correct exception handlers to be invoked (though that structure may be useful for other reasons).” [Miller2002]

2.3. Evaluation and quality metrics

Garcia et al. [Garcia2001] have done a thorough work comparing existent exception handling mechanisms, in regard of their strengths and weakness, as a tool for building dependable object-oriented software.

The most important contributions arising from that work was the definition of a complete taxonomy for classifying the different design approaches to object-oriented exception handling mechanisms, and the definition of a set of quality requirements for the design of

future exception handling mechanisms. Both contributions are useful for the assessment of nowadays and future exception handling mechanisms.

2.3.1. Evaluation

Accordingly to Garcia's study, exception handling mechanisms can be compared in ten different aspects (Table 2.1).

Table 2.1 – Identification of the exception handling models evaluation items.

Evaluation item
Exception representation
External exceptions in signatures
Separation between internal and external exception
Attachment of handlers
Handler binding
Propagation of exceptions
Continuation of the control flow
Clean-up actions
Reliability checks
Concurrent exception handling

The topics on the previous list are discussed on the next sections. Some of these topics have already been addressed, thus, on those cases, and to avoid duplication we will only mention them briefly.

Exception representation

Exception handling mechanisms have different structures for representing exceptions: *symbols*, *data objects*, and *full objects*.

The classic approach is associated with the first kind, symbols. Exceptions are strings or numerical values that are passed to the caller of an operation when an abnormal situation is detected during the execution of that operation. When exceptions are represented by data objects, or full objects, they are created as instances of a class that identifies on exception type (e.g., Java's `IOException`). These exception instances are passed to the exception handler when raised. Exceptions represented by data objects require special support from the programming language, such as a keyword to trigger the exception raising operation (e.g., Java's `throw` statement). Full objects, on the other hand, are just as any other object and have to implement themselves (or some class up in the hierarchy) the

raise operation. Both object representations are preferable to the use of symbols in object-oriented languages. Objects guarantee the uniformity of the programming paradigm; objects do not require the usage of extra global variables for passing information to the exception handler, thus benefiting modularity.

Object-oriented programming languages such as *Ada 95* [ISO8652:1995], *Smalltalk* [Goldberg1989], *Eiffel* [Meyer1988], *Modula3* [Nelson1991], *Guide* [Balter1994], and *Extended Ada* [Cui1992] use symbols to represent exceptions. *Lore* [Caseu1987] and *BETA* [Madsen1993], for instance, use full objects, while other languages like *C#* [ISO23270:2006], *C++* [Koenig1990], *Java* [Arnold2000], *Arche* [Issarny1993], and *Delphi* [Kimmel2001] use data objects.

External exceptions in signatures

Exception handling mechanisms can make the usage of an exception list a mandatory feature or not. In some cases, designers of these mechanisms opt for providing a hybrid solution (e.g., Java). The discussion about the benefits associated with the compulsory usage of an exception list - in opposition to the non-existent or non-compulsory practice - is closely related with the discussion on checked vs. unchecked exceptions (section 2.2.2).

For instance, programming languages like *C#*, *Ada 95*, *Smalltalk*, *Eiffel*, *Delphi*, and *BETA* do not support exception lists. Languages like *C++*, *Lore*, and *Arche* make its usage optional. *Modula3*, *Guide*, and *Extended Ada* use a compulsory approach, while only *Java* allows a hybrid solution (distinguishing checked exceptions from runtime exceptions).

In terms of reliability, exception lists are an extremely helpful feature because they describe all the abnormal responses of a method in its signature. Its compulsory usage guarantees that the programmer is aware of the potential problems that his code will face and is able to provide handlers for those abnormal situations. On the other hand, forcing the programmer to declare all the exceptions that are propagated outside method boundaries, or to handle all the not declared exceptions, can lead to less recommended programming practices such as *exception silencing*, or *log and terminate* for non-fatal exceptions.

Separation between internal and external exceptions

Some systems allow the distinction between *internal* and *external* exceptions. External exceptions are those exceptions that the caller of an operation takes knowledge and has to

handle when the called operation terminates with an unexpected result. Internal exceptions, on the other hand, are not visible to the caller of an operation. They are raised and handled internally and never propagated outside the lexical boundaries that limit the operation. The distinction between these kinds of exceptions can be done, for instance, through the usage of different raising statements: if an exception is activated through the `signal` keyword it will be handled externally; if the exception is activated with `raise`, it is meant to be handled and known only internally.

Currently we are not aware of any object-oriented programming language that implements this separation between internal and external exceptions. Nevertheless, the rules necessary for operating such distinction can be easily verified at compile-time.

Attachment of handlers

The definition of the protected region to which an exception handler is associated can differ in many aspects. For example, a handler can be associated with (i) a statement, (ii) a block of statements, (iii) a method, (iv) an object, (v) a class, or (vi) an exception class. The definition of handlers for a statement or a code block needs special support from the programming language, which must provide keywords and semantics for marking the beginning and the end of the block of code. Associating a handler with a method is the same of associating a handler with a block of code that starts when the method starts and ends when the method ends. A handler can be associated with a particular instance of a class (object handler), thus, each instance of a class can have a different set of handlers. Class handlers allow the definition of the same handling actions for every instance of a class. Handlers associated with exceptions can be activated at any time and anywhere in a program. These handlers are triggered by the raising of an instance of the exception that they are associated with.

Garcia et al. argued against the usage of block handlers. For all possible association kinds, the authors consider this the weakest type. The authors defend that “the use of block handlers violates explicit separation of concerns, since exceptional code is intermingled with normal code albeit moved to the end of the block”. Other authors [Lang1998,Papurt1998] have also shown that, most times, the blocks of statements are defined with the sole purpose of attaching an handler. This practice can lead to the development of software which is difficult to read, maintain and test.

Modula3, Ada 95, C#, C++, Java, Delphi, and Arche programming languages allow the usage of both statement and block handlers. Lore allows statement, class and exception handlers. Guide has statement, method, and class handlers. BETA allows all kinds of handlers associations except block handlers. Eiffel has method and class handlers. Extended Ada uses object and class handlers. Smalltalk only permits class handlers.

Handler binding

On Section 2.2.2 we identified two approaches for discovering the handler that should be executed when an abnormal situation is detected: the *static* approach; and the *dynamic* approach. Garcia suggests that the dynamic approach actually represents two different approaches: the fully *dynamic*; and the *semi-dynamic (hybrid)* approach.

On the static binding approach, handlers are lexically associated with the exception and comply with the lexical hierarchy of the code. On the dynamic approach, the handler for an exceptional event is not known until the exception occurrence. The runtime system inspects the handlers available on the execution stack and chooses the most suitable candidate for treating the exception. The hybrid approach mingles the two previous techniques: a handler can be statically associated with an exception occurrence but, in the event that no suitable handler is found on the immediate lexical context, the runtime system is responsible of dynamically selecting the appropriate handler.

Statically binding a handler to an exception occurrence leads to better readability, it is possible to verify statically which actions will be taken for dealing with a particular event. On the dynamic and semi-dynamic approaches, this kind of analysis are more difficult to perform because binding is dependent of the control flow at run-time. On the other hand, static models do not allow the propagation of exceptions and do not take the call history in consideration when choosing a handler for an exception. Usually, the operation invoker is better suited to handle an exception then the raising code.

Smalltalk, Extended Ada and BETA are examples of programming languages that implement the static approach. Currently, no object-oriented language implements the dynamic model. The hybrid solution is the most common choice for mainstream languages, such as C#, Java, C++, Eiffel, and Delphi.

Propagation of exceptions

Garcia et al. identified two design solutions for exception propagation: *explicit propagation*; and *automatic (implicit) propagation*. The first kind is also known as *single-level* [Liskov1979] propagation because it only allows the propagation of the exception to the immediate caller of the failing operation. However, the raised exception or a new exception can be signaled explicitly from a handler (attached to the caller) to a higher-level component. This kind of propagation is often associated with the static propagation model. In the cases that an exception is not handled locally, it is transformed into a general exception type and propagated to higher-levels or the application is terminated. The second kind of propagation, automatic or *multi-level* [Liskov1979], allows exceptions to be transmitted through multiple levels on the call stack until a suitable handler is found or the program is terminated.

Programming languages implementing static binding (e.g., Smalltalk, Extended Ada, and BETA) do not support any kind of propagation. Most language designers elect the automatic propagation of exceptions as the default behavior on their exception handling models. This technique can be considered unsafe, in terms of exception handling, because it gives no any guarantees that an exception occurrence will be bound to the most appropriate handler. In some cases, explicit propagation can coexist with automatic propagation but, even then, the outcome of the mechanism for handler selection is not entirely predictable. Furthermore, the propagation of an exception through different levels of abstraction (on object-oriented software) can cause the unexpected exposition of implementation details, the degradation of encapsulation and modularity [Yemini1985].

The majority of the object-oriented programming languages use both the explicit propagation and the automatic propagation of exceptions (e.g., C#, Java, C++, Ada 95, and Delphi). Eiffel is a good example of a language that chose to only allow explicit propagation.

Continuation of control flow

As described in the previous section, there are two fundamental propagation models that delineate where the *normal* flow of execution is resumed after the execution of an exception handler: the *termination* (simple and *retry*) and *resumption* models.

As far as we know, all exception handling mechanisms (available in object-oriented programming languages) implement the termination model. Nonetheless, some language

designers, such as the creators of Smalltalk and BETA, decided to implement both models simultaneously. Their decision to include both features is clearly influenced by the power and the flexibility that the resumption model evidences as a programming tool (in some circumstances). Such flexibility comes with a high price: the overall programming model is more complex and, therefore, more error-prone. It is the general understanding that the simpler linguistic and the clearer semantics of the termination model make it preferred, in terms of reliability, to the resumption model.

Clean-up actions

An operation will either terminate correctly or with errors. In both cases, it is important that the program state remains coherent. Clean-up actions allow the program to recover to a valid state, or undo the effects of some actions. There are three design solutions for the implementation of the clean-up mechanism: (i) associated with *explicit propagation*; (ii) usage of a *specific construct*; and (iii) performing *automatic clean-up*. When explicit propagation is used, clean-up should take place in the local handler before the exception is propagated. Using a specific construct, the clean-up is associated with the guarded protected block and will be executed independently of the occurrence or not of an exception. The third solution makes clean-up a platform issue: the system will automatically perform the necessary actions to take the program to a valid state.

Programming languages such as Ada 95, Eiffel, C++ and Archie use explicit propagation. But, the most common approach is the usage of a specific construct, such as the Java's or C#'s `finally` clause.

The most interesting solution in our perspective is making the clean-up actions a platform issue with automatic clean-up. This approach is not currently implemented by any programming language. Thus, in this thesis we will propose a solution for putting into practice this kind of functionality in a transparent and elegant way.

Reliability checks

Some systems implement *static* or *dynamic* checks that test for possible errors introduced by the use of an exception handling mechanism. Yemini and Berry [Yemini1985] suggested several kinds of checks that can be performed by the exception mechanism in order to improve software reliability. Consider, for instance, the following examples: (a) checking the correctness of the parameters used on the exception signaling operations; (b) checking the correctness of the formal set of parameters used on the definition of each exception

handler; (c) checking that only those exceptions that are defined by a signaler are signaled by it; (d) checking that the exceptions are raised and handled in the correct scope. (Note: some of these reliability checks might not be applicable to all mechanisms.)

The first kind of reliability checks are performed by the compiler while the second kind is performed by the runtime system. Ideally, both kinds of checks should be implemented. For assuring a minimal overhead, the larger and more time-consuming set of verifications should be performed statically at compile time. Most object-oriented programming languages and platforms already implement this dual version (e.g., C#, Java, Eiffel, Delphi, and Guide). Ada 95 and Smalltalk are examples of languages that only implement dynamic checks.

Concurrent exception handling

The support for concurrent programming within an exception handling mechanism can be classified in three distinct ways: (i) *unsupported*; (ii) *limited*; and, (iii) *complete* [Garcia2001]. The first kind is the total absence of support for concurrent activities in the mechanism. Systems with limited support provide the means for notifying all the threads involved in a computation of the occurrence of an exception. Thus, the exception can be handled by multiple threads simultaneously and in different ways. A system with complete support for concurrent exception handling provides the facilities to combine the concept of atomic actions with concurrent exception handling. Such system usually implements the means to allow the final synchronization of all the participants in an action, the resolution and propagation of an exception through multiple threads, and the invocation of the different handlers associated with each of the action participants.

2.3.2. Quality requirements

Garcia et al. [Garcia2001] proposed a criteria for guiding the design of an effective exception mechanism for object-oriented software development. The authors have an extensive experience designing and developing dependable object-oriented systems and exception handling mechanisms. Based on their collective experience, they defined a well structured list of guidelines to help developers create the best possible exception handling mechanisms in terms of software reliability. These guidelines are spawn over twelve items: Readability; Modularity; Maintainability; Reusability; Testability; Writeability; Consistency; Reliability; Simplicity; Uniformity; Traceability; and Performance.

Q1. Readability – Exception handling mechanisms must provide for better code readability in programs that have to deal with (and recover from) errors or any other kind of abnormal event. It is consensual that software complexity has, undoubtedly, grown with the advances in systems and development tools. The number of exception types and instances that are possible to find in modern programs is just overwhelming. Furthermore, automatic code generation tools are becoming popular and, for these specific source code producers, size is not an issue. To increase code readability it is important to allow a clear separation between *exceptional* code and *normal/business* code. Systems must provide for: (a) a clear definition (identification) of the protected regions of code; (b) an easy recognition of the exception handling code, its boundaries and control flow behavior.

Q2. Modularity – An important characteristic of object-oriented systems is encapsulation. An effective exception handling mechanism must be able to maintain object encapsulation as much as possible. For that effect, exceptions occurring inside a component must be confined to that component or propagated to immediate neighbors in a controlled manner. Avoiding the unnecessary propagation of exceptions outside the raising component's borders prevents the disclosing of internal implementation details and complies with the information hiding policy inherent to every object-oriented system.

Q3. Maintainability – In 1988, Bertrand Meyer stated on his book “Object-Oriented Software Construction” [Meyer1988] that 70% of software cost is directly related with software maintenance. Both the maintenance costs and the probability of introducing new errors during maintenance tasks increase with software complexity. To prevent causing more harm than good when performing such actions, exception handling mechanisms must be projected for simplicity and readability, thus making software easy to modify and/or correct.

Q4. Reusability – One of the main features of object-oriented software is component reusability. Reusability is achieved through good software design and a clear separation of concerns among system components. Reusability of both business and exceptional code is enhanced by a straight separation between exception handlers' code and business code.

Q5. Testability – The purpose of testing is exposing the potential faults in systems, thus avoiding their subsequent manifestation at run-time. The existence of multiple execution paths, due to the introduction of exception handling code, makes software testing harder and more complex. It also raises the problem of achieving full code coverage in tests (it can be difficult to test the code inside exception handlers). Object-oriented software testing is

still an evolving area with many concerns to be addressed [Pezz2004]. When testing, it is important to verify that each exception is being correctly handled, and that the exception handling code does not introduce new errors. While projecting a new exception handling mechanism, designers must have in mind that their structural design must not impose further weight on the testing procedure.

Q6. Writeability – For the programmer, the task of writing error recovery code can be difficult and complex. For the programming language designer, it is also hard to control the emerging complexity of his language. Some systems, due to their nature, size or complexity, may require more expressive language constructs or even the means to separate different error handling policies throughout different applications levels. Occasionally, too much expressive power can be counter-productive and lead to the creation of code that is harder to read.

Q7. Consistency – For achieving maximum dependability in object-oriented systems, it is mandatory keeping each system component in a consistent state, even in the presence of errors. To prevent catastrophic failures, systems must be able to continue their execution in a consistent state even if a component fails to complete a requested operation.

Q8. Reliability – Exception handling mechanisms are, by nature, tools to construct reliable systems. Thus, it is capital that the mechanism itself is error-free. Such reliability level is achieved (among other reasons) through exhaustive testing, careful design, a correct expressiveness level, and complete integration with the programming language and/or framework. By themselves, the mechanism and the exception handling code must not be the source of new failures. The system has got to provide means to permit the verification and testing of the error handling code.

Q9. Simplicity – The task of writing exception handling code ought not to impose itself over the task of writing the application's business code. Many critical systems developers may not be completely sympathetic with this point of view but, ultimately, the truth is that the business code is the core responsible for providing the expected system functionality while the exception handling code is responsible for avoiding the program's premature or invalid termination. To let developers concentrate on the writing the application's business code, exception handling mechanisms should not increase the overall system complexity. Exception handling code must be easy to write, maintain and read. Thus it must be as simple as possible, have clear semantics, and its behavior must be well defined for all execution scenarios.

Q10. Uniformity – The syntax and the concepts introduced by the exception handling mechanism cannot violate the object-oriented nature of a system. Abstraction, encapsulation, modularity, and inheritance must not be *broken* by the establishment of an exception handling system. Failing to fulfill this requirement causes a serious handicap in terms of software reusability, modularity and testability. Moreover, the mechanism's constructs should stick to consistent syntactic conventions and should not provide multiple representations for the same concept.

Q11. Traceability – The exception handling mechanism ought to provide the means to assess the nature of an abnormal occurrence and deal with it. In both cases, it is necessary to gather all the available information about the exception and pass it to the right entity (handler). The exception handling systems must provide the means to communicate the name, description, location, and severity of an exception (and further relevant information) to its handler (or handlers).

Q12. Performance – Performance is traditionally a major concern in systems design. The exception handling mechanism should not undermine the overall system's performance by introducing unnecessary overhead. Language designers usually try to ensure the best possible performance from two different perspectives: (a) allowing systems, either using exceptions or not, to offer equal response times if no abnormal occurrences are detected; (b) minimizing the time spent searching for the suitable handler to deal with an exceptional event. Nonetheless, the prime concern of an exception handling system should be to provide the means to implement error handling while allowing fast recovery.

2.4. Backward error recovery

Exception mechanisms can use both *forward error recovery* and *backward error recovery* strategies for dealing with occurring exceptions. The aim of a *forward error recovery* mechanism is to move the system into a correct state using knowledge about the current erroneous state. The state recovery actions are application-specific by their nature and are based on correcting or isolating the effects of a fault. This allows the normal operations to be continued. Forward error recovery techniques are useful for handling anticipated faults, which can be detected and abstracted as exceptions. *Backward error recovery*, on the other hand, returns the system to a previously consistent state (saved before the failure manifestation). The techniques used for accomplishing this objective are, typically, application-independent, transparent for the application (e.g., atomic transactions and

```
ensure <acceptance test>
    by <1st (primary) alternate>
    else by <2nd alternate>
    else by <3rd alternate>
    .
    .
    .
    else by <nth alternate>
    else error
```

Listing 2.6 – The notation for a recovery blocks structure

checkpoints). Such techniques involve the rollback of the system state and undoing of the effects of the computation performed since that state. Backward recovery techniques are broadly applicable in dealing with unanticipated error conditions.

The *recovery block* construct [Horning1974] is a program-controlled backward error recovery technique used in sequential programs. It provides mechanisms for specifying recovery points, acceptance tests, and alternate program code for execution.

Horning et al. proposed the notation for the recovery blocks mechanism in 1974. This language construct allows developers to define tests of acceptability and correctness on intermediate stages of execution of the program and also to declare alternative courses of action should the tests prove negative. Recovery blocks have been described by Randell as a structure for software fault tolerance [Randell1975] and by Anderson as “a proof-guided methodology for constructing the checks for acceptable program behavior” [Anderson1975].

Listing 2.6 illustrates the usage of a recovery blocks structure. The acceptance test¹ yields a logical value and must not have any side effects. Its evaluation allows the developer to verify the acceptability of some condition. Each alternate is a block of code with finite set of statements. The execution of an alternate is dependent of the last logical value yielded by the acceptance test. The test is first performed when execution reaches the recovery blocks structure, and the execution of the first alternate is dependent of the value yielded at that point (if the test proves negative the first alternate is executed). The acceptance test is then performed each time an alternate concludes its execution (error-free) and, in the case that the test result as not changed, the next alternate is executed. A fundamental feature of this technique is the setting of the program state to what it was on entry to the recovery block

¹ The writing of good acceptance tests is a complex and error-prone task. It is considered the major shortcoming of this technique.

before the execution of an alternate. The execution of the recovery block ends when the acceptance test yields “true” or there are no more alternates, in which case an error condition is raised externally to the (concluded) recovery block and any further recovery can only be performed by an enclosing recovery block. If there are no more recovery blocks, the system terminates the program. In terms of error handling, a failed acceptance text means that an erroneous condition has been generated (raise). In some cases, acceptance tests might be unable to detect *internal errors*¹. In these situations, in the presence of internal error conditions, the acceptance tests may still yield “true” and therefore acceptable results. Internal errors can also arm the information structures upon which the recovery mechanism operates.

The intent underlying the execution of each alternate is satisfying the acceptance test. This does not mean that each acceptance test implements the same functionality, in fact, it is quite the opposite. The lesser the degree of similarity between two alternates, lesser are the chances of both sharing a common design inadequacy.

Early systems implementing the recovery blocks mechanism relied on a *recovery cache* [Horning1974] for performing the state restoration between alternates execution. The recovery cache approach offered certain advantages over conventional *checkpointing* techniques. It assured that only the values affected by the test and alternates execution are preserved for the life-times of the appropriate recovery blocks. Furthermore, the preservation and reinstatement of these values was completely automated and not susceptible to human errors of omission.

The usefulness of recovery blocks rests on the effectiveness of the acceptance tests on detecting abnormal end error conditions. If the acceptance test is something more than the “*most simple as possible*” there will be a significant chance that it will itself contain design faults. In such cases, tests can fail to detect errors or give origin to *false-positive* results. Moreover, too complex acceptance tests can be the source of unacceptable run-time overhead. Developing simple, objective, and effective tests can thus be a difficult and potentially harmful task for the program’s overall reliability.

Acceptance tests should not be the only means of error detection. For instance, assertions and hardware run-time checks should be on the front line of error detection. Any exceptions raised inside an alternate will activate the same recovery action as for

¹ Errors that are not pure algorithmic but due to the violation of the machine specifications and can cause *normal* systems to abandon execution.

acceptance test failure. Thus, each alternate should implement its own fault-tolerance mechanisms and, in the cases where this does not suffice, an exception is raised to notify the run-time mechanism that the recovery block was unable to accomplish its objective and execution should be handled by upper-level recovery blocks (if it exists).

Besides recovery blocks, the best known technique based on design diversity is probably *N-Version programming* [Avizienis1977]. N-Version programming is a software diversity technique in which all the versions are designed to satisfy the same requirements and output. Correctness is based on the comparison of all the outputs. In contrast with the recovery blocks approach, this technique uses a generic decision algorithm (usually a voter) to select the correct output. Creating different algorithms to achieve the same end-goal does not necessarily mean that development is more complex than for a single version. Nevertheless, it requires substantially more development time and effort. The design diversity is useful to minimize the probability that two or more versions will produce similar erroneous results for the same *decision action* (e.g., voting). The support for N-Version software is provided by a dedicated execution environment, which implements the decision algorithms and all the necessary means for the execution of the versioned-software.

Other multi-version software approaches for fault-tolerance were developed based on the Recovery Blocks and N-Version programming techniques. This is the case of N Self-Checking programming [Laprie1987,Laprie1990,Laprie1995a], that uses separate acceptance tests for each version. The Consensus Recovery Blocks [Scott1987] approach combines N-Version Programming and Recovery Blocks to improve the reliability. And, the $t/(n-1)$ -Variant Programming [Xu1997] that implements a different output selection mechanism that guarantees that at least one non-faulty execution unit exists.

2.5. Real-time concerns

In our work, we do not address any issues related with real-time systems. Nevertheless, we feel that it is important to have a look into the difficulties that such systems impose, in terms of exception handling, in order to provide a thorough perspective on the current state of art.

The truth is that exception handling mechanisms for real-time systems have never been a priority for programming language designers. The execution principles behind real-time

software have always been, in a sense, prohibited for any type of functionality that might raise uncertainty about the time necessary to complete an operation [Lang1998].

A real-time system¹ has to comply with predetermined execution deadlines. Unfortunately, traditional exception handling mechanisms do not offer any time-related guarantees. They can introduce overhead in the execution of normal code and do not allow the predetermination of exception handlers execution times. Thus, when an exception is raised, there is no efficient way of predicting what will be the delay introduced by the exception handling system or by the recovery actions. In real time systems, it would be acceptable to have longer startup times in order to allow an application to offer time-bounded detection and handling of exceptions. Unfortunately, no current performance optimization design trend (as we have seen in Section 2.3.2) follows such direction. The most common optimization effort is on having applications exhibit the same response times under *normal* system operation, either when an exception mechanism is being used or not.

Real time processes and threads are usually scheduled in accordance with their priorities. A lower priority process cannot preempt a higher priority one. On the other hand, a higher priority process can preempt a lower priority execution, if necessary. The same should be true with exceptions code: a lower priority exception handler should never preempt a higher priority execution. Furthermore, exceptions should also have priorities associated, in a way that higher priority exceptions should be handled prior to their lower priority counterparts when multiple occurrences are detected simultaneously.

In an ideal real-time system, the overhead imposed by the definition of exception types, exception handlers, exception detection and handling, should be tuned in order to allow the application to execute within pre-established time constraints, either under *normal* or *exceptional* execution. Compilers and automatic testing tools could play an important role in this respect.

2.6. Other approaches

The exception handling mechanisms described so far represent, what we consider to be, the more relevant advances in terms of exception handling in recent history. Nonetheless,

¹ In Hard-real-time systems, it is a fatal error for a function not to attain its time constraints. In soft-real-time systems, having a function not meeting its time constraints is a serious problem but not fatal.

there have been other important efforts for improving systems resilience to exceptional events that are worth mentioning.

2.6.1. Aspect Oriented Programming

The current trend in the design of exception handling mechanisms is making systems that impose some kind of separation between the code for implementing different aspects of a program. For instance, in [Lemos2001], Lemos and Romanovsky propose an approach that separates the handling of requirements-related, design-related, and implementation-related exceptions during the software life cycle.

With the advent of Aspect Oriented Programming (AOP) [Kiczales1997,Elrad2001] a new approach was introduced. AOP is a programming paradigm that increases modularity by allowing the separation of *cross-cutting* concerns. In the object-oriented paradigm, different concerns are grouped (packaged) inside methods, objects, classes, and packages. But, in some cases, there are concerns that are transversal to such kinds of packaging and/or do not share a hierarchical relation, thus are called *cross-cutting* concerns.

Exception handling has been considered a potential application area for AOP since its origin in the early 1990's at the Xerox Park [Kiczales1997]. Some authors have suggested that the AOP approach can be used to separate exception handling code from business-logic code [Lippert2000], proposing the treatment of exceptional behaviors as a cross-cutting concern to the application. Additionally, this approach can contribute to increase the readability of both *normal* and *exceptional* code, and to avoid the mingling between both kinds of code by allowing the developer to focus on each one of the following tasks separately: writing the exception handling code and writing the business logic code.

Martin Lippert and Cristina Lopes [Lippert2000] have shown how an AOP tool for Java can be used to modify an application source code to apply exception handling and detection as a crosscutting concern. The exception detection and handling code of the JWAM [JWAM2008] framework were partially re-engineered using AspectJ [Lopes1998], an AOP extension to Java. AspectJ was essential for re-writing the framework's source code. The authors collected information from the source code, pre and post reengineering, to analyze the advantages and disadvantages of the process.

Rewriting JWAM's code using AspectJ represented a cut of $\frac{3}{4}$ of the original exception handling code. Most of the redundant code (imposed by the programming language) was eliminated using AOP. The experiment demonstrated that the code that deals with

exception detection and handling can be substantially reduced if applied in a cross-cutting manner. Furthermore, using AOP to develop exception handling code increases code reusability, makes the code clearer, provides better support for different configurations, better tolerance for changes in the specifications, and better support for incremental development. On the other hand, the authors reported that it is difficult to reconstruct the local effects of the insertion of aspects into the code. The AOP tools available do not provide sufficient support and do not allow seeing the “whole picture” in a convenient manner. The developer is forced to browse through several source code files if he or she wishes to understand the complete functionality associated with a single location in the program.

Filho et al. [Filho2006,Filho2007] are strong supporters of using AOP to introduce exception tolerance capabilities into a program. They suggest lexically separating error-handling code from normal code so that both code types can be independently implemented by different developers and modified separately. In addition, they propose leveraging AOP to enhance the separation between error-handling code and normal code.

2.6.2. Exception handling for Futures

Zhang et al. [Zhang2007a] defined a *future* as “a simple and elegant construct that programmers can use to identify potentially asynchronous computation and to introduce parallelism into serial programs”. Several languages implement the *futures* mechanism [Halstead1985,JSR166,Charles2005,Zhang2007a] and implementations vary in terms of syntax and semantics. For instance, some systems provide a list of interfaces that can be used to *mark* classes or methods in serial programs as *futures* (able to execute concurrently) [Halstead1985] while other systems use annotations to identify the same kind of computations on the code [Zhang2007a].

Futures represent an extremely valuable mechanism for creating concurrent applications in a simple way, either from the ground-up or by transforming serial programs into new parallel versions. But, although *futures* make concurrent programming a much simpler task, they also introduce new difficulties in terms of exception handling.

```

public class Fib
{
    public int fib(int n) {
        if (n < 3) return n;
        @future int x = fib(n-1);    //declaration
        int y = fib(n-2);           //point of usage
        return x + y;
    }
    ...
}

```

Listing 2.7 – Futures utilization within the *DBLFutures* framework

With *futures*, it is possible, for instance, to start a parallel computation of a variable value and continue the main execution until the value is first used. When the main computation reaches this point, either the value is ready to be used or the program waits for the completion of the execution of the parallel thread (Listing 2.7). The *future* value can be computed asynchronously independently of the location where it is defined in the code. This allows the system to organize the code in several distinct blocks of code that can be executed concurrently.

In terms of exception handling, *futures* introduce uncertainty about where is the correct location to handle an exception raised by a *future* value computation. Exceptions can either be handled in the serial way, at the same location where the future is first declared, or at the point where the future value is used. On the later case, the exception is kept inside the *future* and delivered to the calling code when the *future* return value is requested¹.

There are two different approaches for dealing with exceptions in the *DBLFutures* [Zhang2007a] (and in all *futures* mechanism implementations in general). The code snippets in Listing 2.8 illustrate both of them. Method `f1()` returns the sum of variable `x` and `y`. Variable `x` holds the value returned by a call to `A()` and `y` the value returned by a call to `B()`. Variable `x` is declared as a *future*, thus `A()` can be executed concurrently, independently of the place where it is invoked in the code, even if the code is written in a serial form. In *a*) any exception raised in `A()` is delivered to the same point that it would be delivered if the program executed sequentially. This is what the authors call the *as-if-serial* exception handling mechanism. On the other hand, in *b*) exceptions will be delivered to the point where the *future* return value is used. Just looking at the examples, we can conclude that the second implementation, if depleted of *DBLFutures* semantics, is incorrect for

¹ In Java 5.0 Future APIs, exceptions from future execution are propagated to the point in the program at which future values are used.

<pre> public int f1() { @future int x; try{ x = A(); }catch (Exception e){ x = default; } int y = B(); return x + y; } </pre> <p style="text-align: center;">a)</p>	<pre> public int f1() { @future int x; x = A(); int y = B(); try { return x + y; }catch (Exception e){ return default + y; } } </pre> <p style="text-align: center;">b)</p>
---	---

Listing 2.8 – Examples of exception handling in *DBLFutures*

compilation in its serial form, while the first version remains correct. Furthermore, the first approach provides programmers with more intuitive understanding of the exception handling behavior and control, while the second one can be harder to read and less intuitive.

To allow the employment of true *as-if-serial* semantics for exception handling in futures, the authors in [Zhang2007a] suggest that it will be necessary to resort to a Software Transactional Memory (STM) mechanism [Shavit1995] in order to ensure that the global side effects of parallel execution of a program (using futures) is consistent with that of the serial execution. As far as we know, no current approach truly implements the *as-if-serial* semantics for exception handling using STM.

The combination of the *futures* mechanism with the *as-if-serial* semantics for exception handling turns the development of concurrent programs into a much simpler and straightforward task than in the past. We risk saying that the development software for multi-core (multi-processor; or cluster) architectures in the future will undoubtedly be linked with the advances in the support for the *futures* mechanism in the mainstream programming languages. Fortress [Allan2005] and X10 [Charles2005] are two programming languages, currently under active development, that are expected to simplify the way concurrent applications are written. Both languages have constructs similar to *futures*.

2.6.3. Compensation stacks

Weimer et al. in their work on “Exceptional Situations and Program Reliability” [Weimer2008] describe two interesting and self-complementing ideas that can help improve the reliability of programs: (1) a mechanism that allows an *in-deep* analysis of a

program code for finding all defects that can lead to resource-handling failures in exceptional situations; (2) based on this work finding defects, the authors propose a language feature, named *compensation stacks*, for ensuring that simple resources and API rules are handled correctly even in the presence of run-time errors.

Weimer et al. describe a static data-flow analysis for finding program defects. This analysis uses a *fault model* and a *formal specification* of proper resource handling to guide the defect detection process. As an output, the analysis creates a *defect report* that includes a program path, one or more run-time errors and one or more resources governed by the specification. In broader terms, if a run-time error occurs at any point identified in the report, the program is prone to violate the specification for the resources in use. A fundamental aspect of this analysis is the fact that it considers not only the code in the *normal* control flow but also the control flow related to the exceptions in the fault model.

The main goal of the Weimer analysis is to identify a path, with its origin in the start of method and ceasing at the end of the same method, where a resource is not in an accepting state. This process requires the construction of control flow graph accordingly to the pre-defined fault model as well as the formal specification. The specification is responsible for describing what the program must do and the fault model will describe what can go wrong. To formalize the way how programs should manage and use certain resources and interfaces, the authors propose the utilization of Finite State Machine diagrams diagrams (FSMs) [Ball2001,Deline2001].

Weimer et al. alerted that their technique may “spuriously report correct code as having defects and may fail to report real defects”. Nevertheless, they declare to have obtained no false positives in their experiments while covering over five million lines of code and having found over 1300 defects.

Based on the results encountered during the previous experiments, the authors claim that “try-finally blocks are ill-suited for handling certain classes of resources in the presence of run-time errors. (...) In essence, however, exceptions create hidden control-flow paths that are difficult for programmers to reason about”.

Features existing in programming languages, such as *destructors* and *finalizers*, can assist programmers in the task of preparing their code for correctly dealing with resources in the presence of run-time errors. Destructors provide guaranteed cleanup actions for stack allocated objects even in the presence of exceptions because they are tied to the dynamic

call stack of a program in the same way that local variables are¹. A finalizer has a different way of assuring the correct release of resources, it executes on an instance of a class only when that instance is about to be reclaimed by the garbage collector. The garbage collector gives no guarantees about which instances will be reclaimed, the order they will be reclaimed or the time-frame for the operation. Finalizers perform, in a certain way, a kind of *lazy* clean-up.

Weimer and colleagues propose a new approach for assuring the execution of resource cleanup actions even in the presence of exceptions, the *compensation stack*. This language feature is influenced by the concepts of compensating transactions² [Korth1990], linear sagas³ [Alonso1994,GarciaMol1987], and linear types [Deline2001] to create a model in which obligations are recorded at run-time and are guaranteed to be executed along all paths.

Compensation stacks are seen as a kind of a generalized destructor that can be used to execute arbitrary code and not just to invoke functions upon object destruction. Based on the fact that many program actions require that multiple resources are handled in sequence, the compensation stack system links actions with compensations, and guarantees that if an action is taken, the program will not end without executing the associated compensation. Compensation stacks deallocate resources based on lexical scope, much like a destructor do, but they are also first-class objects that make use of finalizers to ensure that their contents are eventually executed.

A compensation stack contains several *closures*. Closures are run automatically (in a *last-in, first-out* order) when a stack-allocated compensation stack goes out of scope or when a heap-allocated compensation stack is finalized. Nevertheless, programmers are still free to arbitrarily push closures onto compensation stacks and run closures ahead of time. This is important to allow the usage of an ordinary programming idiom where resources must be freed as early as possible along each path. Upon termination of the execution of a

¹ The programmer must still remember to explicitly delete heap-allocated object along all paths.

² A compensating transaction semantically undoes the effect of another transaction after that transaction has committed.

³ A saga is a long-lived transaction seen as a sequence of atomic actions $a_1 \dots a_n$ with compensating transactions $c_1 \dots c_n$. Either $a_1 \dots a_n$ executes or $a_1 \dots a_k c_k \dots c_1$ executes. Note that the compensations are applied in reverse order.

compensating action, either normally or exceptionally, the compensation is eliminated from the compensation stack.

Another interesting argument about compensation stacks is the fact that the system ensures that the execution of compensation will continue even if a compensating action raises an exception during its execution (the exception is simply logged).

Compensation stacks provide more flexibility than standard language approaches to adding linear types or transactions by moving bookkeeping from compile-time to run-time and enforcing a certain ordering on the execution of compensations.

2.7. Summary

Early error detection and handling mechanisms, such as *error codes* and *status flags*, were proven to be insufficient for dealing with all possible abnormal situations and with the overall increase in the complexity of programs. The exception handling model emerged as a first effort to regulate the way programs deal with exceptional situations.

Exceptions eliminate many of their ancestors' shortcomings. From our point of view, the most important contributions of the exception handling mechanism in terms of reliability are:

- Preventing any abnormal or erroneous situation of passing undetected, thus avoiding the continuous execution of a program in a corrupted state or based on false premises;
- Providing the means for a programmer to plan a set of handling actions for dealing with abnormal occurrences at location, thus allowing the retrying of a failed operation or its replacement with a working version;
- Allowing the communication of a error condition to different locations in the program in a structured fashion;
- Improving error handling in distributed systems;
- Increasing code readability.

Unfortunately, such improvements have a cost - the overall complexity of systems and their code will increase. The original Goodenough's specification [Goodenough1975] has

evolved and been extended originating different flavors of the exception detection and handling mechanism.

The main factor of distinction between modern exception handling mechanisms is the way execution flow is continued after an exceptional occurrence. Existing systems can implement the *termination* model, the *retry* model, or the *resumption* model. In some cases, the same system can provide more than one way of dealing with control flow. We have attested the positive and negative features of each model and concluded that each one excels in different scenarios. Nonetheless, the termination model continues to be the preferred option for the great majority of programming language designers, due to its overall simplicity and clear semantics. In addition, we can state that it also covers most situations that occur in practice and, under specific circumstances, it can mimic the remaining models.

Exceptions provide better means for classifying errors and abnormal occurrences. It is fairly straightforward to create classes representing exceptions, to derive sub-types and create *class-like* exception hierarchies in order to fine tune the exception identification process. In general, most systems deal with occurring exceptions independently of who raises the exception. But, some models allow the programmer to *bound* exception handling actions to exception occurrences inside specific components, other can even make exception handling *conditional* and impose that several *pre-conditions* are met before entering an exception handler.

Some exception models allow the declaration of the exceptions being raised inside a component on that component's interface. This is, simultaneously, a way of civilizing exception propagation between application components and publicly declaring which exceptions might occur during on a certain function call. Components missing an *exception list* will require solid and thorough documentation of its exceptions in order to alert programmers for their hidden hazards. On the *checked vs unchecked* exceptions discussion, such feature is also a point of rupture. Moreover, we can say that checked models give privilege to the obligation of detecting, handling and communicating occurring exceptions, while unchecked models, on the other hand, give privilege to the writing of normal application logic code. Checked models allow the compile-time checking of the completeness of the exception handling code. Unchecked models, on the other hand, require exhaustive testing in order to assess the complete coverage of the error handling code.

Garcia et al. [Garcia2001] proposed a criteria to evaluate the quality of an exception handling model in terms of reliability. At the same time, they were also able to provide a set of quality metrics to help guiding the development of future exception handling models. Among the programming languages currently better suited in terms of exception handling, using the Garcia classification, are Guide and Java.

Along this chapter we mentioned several problems or difficulties associated with the usage of exception handling mechanisms. These problems are summarized on the following list:

- It can be difficult to use testing techniques to find defects and evaluate programs' behavior in exceptional situations [Sinha1999, Malayeri2006]. For instance, coverage metrics tests require previous knowledge of the implicit control flow on an exceptional situations and to validate exception handling code one might have to use fault injection techniques;
- Building a complete list of the exceptions that are prone to be raised by a component, prone to be raised at specific location in the code, or knowing the origin of the identified exceptions, and their propagation path is an overwhelming task;
- Language-level exceptions introduce implicit control flow which can mine software reliability;
- Exception handlers are usually lexically scoped and might be quite labyrinthic ;
- Handling failures from multiple resources in a location where they are used lexically close one to another is difficult and a potential cause for code errors;
- Handling multiple cascading exceptions (nested protected blocks and handlers) can lead to serious program defects;
- The inclusion of exception handling code into the application logic code of a program increases the distance between a resource allocation, its usage, and its consequent release;
- Goodenough [Goodenough1975] proposed that exceptions should not be used only on rare occasions. Nowadays, programming languages, software libraries, operating systems, execution platforms, middleware and development frameworks, all declare and use different types of exceptions. The number of

different exceptions existing in a medium-size application can reach impressive numbers, in the order of thousands. Dealing with all exceptions types (and subtypes) is a cumbersome, complex and error-prone task. The option of handling none is unfeasible in terms of reliability. Handling only a smaller part might not be sufficient.

We can safely conclude that existing exception handling mechanisms, if correctly used, are a great tool for improving software reliability. Exception handling is, by far, the most popular reliability mechanism in use in modern programming languages. Regrettably, the overwhelming complexity of dealing with all possible abnormal situations is making exception handling less attractive for programmers, and, in the end, making programs less reliable.

In the next chapter, we will show how the problems just mentioned are affecting the way programmers use exception handling mechanisms and how that can be a reason for concern in terms of the overall software reliability. As we mentioned in the introduction of this chapter, sometimes programming language designers are forced to go back to the design table and adapt their models in order to comply with the way users relate with them.

We believe that a possible solution for this problem is making the developers' task simpler. It is necessary to create smarter tools. Tools that can, for instance: help detecting exceptions before run-time; help creating the code for handling exceptions; automatically verify handlers' code during testing; among others.

A Field Study in Exception Handling

Most modern programming languages rely on exceptions for dealing with abnormal situations. Although exception handling was a significant improvement over other mechanisms like checking return codes, it is far from perfect. In fact, it can be argued that this mechanism is seriously limited, if not, flawed. This chapter aims to contribute to the discussion by providing quantitative measures on how programmers are currently using exception handling. We examined 32 different applications, both for Java and .NET, and, by doing so, we were able to conclude that exceptions are not being correctly used as an error recovery mechanism.

Another aspect taken into consideration when reasoning about the efficiency of exception handling code in programs is the quality of the existing documentation for exceptions. For years, programmers trusted in the correct documentation for error codes returned by procedures to correctly handle erroneous situations. Now, they have to focus on the documentation of exceptions for the same effect. In the second part of this chapter, we show to what extent can exception documentation be trusted and how it tends to be scarce. This study provides a useful quantitative measure for guiding the development of new error handling mechanisms.

3.1. Introduction

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

Since the appearance of exception handling mechanisms their importance has been steadily increasing. Being a part of modern object-oriented programming languages such as Sun's Java [Sun2006] and Microsoft's .NET [ISO23271:2006], exceptions have been slowly replacing the *error codes* that are widely used in procedural languages like C. Nonetheless, the exception handling mechanism is far from perfect. Problems include¹:

- Due to the large amount of existing exception types (and their subtypes) in modern software, programmers tend to use and throw generic exceptions, making it almost impossible to properly handle errors and recover for abnormal situations without shutting down the application;
- Programmers are also prone to catch generic exceptions, not providing proper error handling, and making the programs continue to execute with a corrupt state (e.g., in Java). On the other hand, in some platforms, programmers do not catch enough exceptions making applications crash even on minor error situations (e.g., in C#/.NET);
- On the other hand, programmers that try to provide proper exception handling see their productivity seriously impaired. A task as simple as providing exception handling for writing a file to disk may imply catching and dealing with tens of exceptions (e.g., `FileNotFoundException`, `DiskFullException`, `SecurityException`, `IOException`, etc.). As productivity decreases, cost escalates, programmer's motivation diminishes and, as a consequence, software quality suffers;
- Due to the semantics and expressiveness level imposed by programming languages onto exception constructs, providing proper exception handling can be

¹ Please, refer to Chapter 2 for a more detailed discussion on the mechanism shortcomings

quite a challenging and error prone task. Depending on the condition, it may be necessary to enclose `try-catch` blocks within loops in order to retry operations. In some cases it may be necessary to abort the program or perform different recovery procedures. Bizarre situations, like having to use nested `try-catch` blocks to deal with an exception while trying to close a file on a `catch` or a `finally` block, are common. Dealing with such issues correctly is quite difficult, error prone, not to say, time consuming.

To make things interesting, the debate about error handling mechanisms in programming languages has been refueled with the launch of Microsoft's .NET platform. Currently, the Java Platform and the .NET platform constitute the bulk of the modern development environments for commercial software applications. Curiously, Microsoft opted to have a different exception handling approach than Java. In .NET the programmer is not forced to declare which exceptions can occur or even deal with them. Whenever an exception occurs, if unhandled, it propagates across the stack until it terminates the application. On the other hand, in Java, in most cases, the programmer is forced to declare which exceptions can occur in its code and explicitly deal with exceptions that can occur when a method is called. The rationale for this is that if the programmer is forced to immediately deal with errors that can occur, or re-throw the exception, the software will be more robust. This way the programmer must be constantly thinking about what to do if an error occurs and acknowledge the possibility of errors.

On the .NET's camp, the arguments for not having checked exceptions that are normally used are [Gunnerson2000]:

- Checked exceptions interfere with the programmers' productivity since they cannot concentrate in business logic and are constantly forced to think about errors;
- Since the programmer is mostly concentrated in writing *business logic* and not dealing with errors, it tends to *shut-up* exceptions, which actually makes things worse. Corrupt state is much more difficult to debug and correct than a clean exception that terminates an application;
- Errors should be "exonerated" by exhaustive testing. A sufficiently accurate test suite should be able to expose dormant exceptions, and corresponding abnormal situations. For the problems that remain latent, it is better that they appear as a

clean exception that terminates the application than having them being swallowed in a generic catch statement which leads to corrupt state.

Obviously, both camps cannot be 100% right. But, overall, the important message is that in order to develop high-quality robust software, in a productive way, new advances in error handling and new perspectives into the subject are needed. Our work aims to contribute to the discussion by providing quantitative measures on how programmers are currently using exception handling. This chapter aims to contribute to the discussion by providing quantitative measures on how programmers are currently using exception handling. We targeted in particular the .NET and Java platforms, as well as the C# and Java programming languages.

The use of unchecked exceptions, more precisely the difficulty that their usage introduces due to the lack of an *exception list* mechanism when a programmer needs to know what exceptions a method call may raise, increases the importance of good exception documentation. In our days, if a programmer expects his code to be used by others, or even himself, he or she has to spend time and effort documenting his methods, explaining the circumstances in which a given erroneous situation can occur and how methods act on those events. This documentation process, being mostly a manual one, is subject to incompleteness and faults. The absence of good code documentation is bound to cause programming errors, because unless there is a way of examining the source code of the software module a programmer is interacting with, documentation is all he or she can trust.

These problems are known by programmers who spend precious time wrestling with bad code documentation, and especially by those used to the older way of error code identifiers. Nowadays, because error codes can still be used, for some developers exception handling is an optional way of dealing with erroneous situations [Ryder2003]. Consequently, the full impact of the problem of poor exceptions documentation in modern programming languages is not known. We propose to assess the impact of such problem by analyzing the code and the documentation released with a set of open-source programs.

To our knowledge, this is the most comprehensive study done to date on exception handling. The data presented on this chapter is important to guide the development of new mechanisms and approaches to exception handling.

3.2. Programming with exceptions

To understand the way programmers use exception handling constructs in modern programming languages, we have to look into both the source code and the executable files.

We examined 32 different applications, both for Java and .NET, covering 4 different software categories: libraries; stand-alone applications; servers; and applications running on servers. Overall, this corresponds to 3 410 294 lines of source code of which 137 720 are dedicated to exception handling. For this work, we have processed 18 589 try blocks and corresponding handlers.

In this section, we assess the usage of exception handling code in the targeted programs regarding the following topics:

- a) percentage of exception handling code;
- b) the type of actions performed inside exception handlers;
- c) the classes used as exception handler arguments;
- d) the exception types most frequently caught;
- e) the call stack levels that an exception travels before it is caught;
- f) the size of handlers;
- g) the types of handlers;
- h) the usage of checked or unchecked exceptions;
- i) the usage of a retry-like functionality.

3.2.1. Methodology

Selecting an adequate set of applications for processing was quite an important step. It was necessary to guarantee that both the source code and the binaries of the applications were available. The source code of each application had to be representative of common programming practices for the target platforms. Also, care had to be taken so that these would be “real world” applications developed for production use (i.e., not simply prototypes or beta versions). This was so in order not to bias the results towards immature

applications where much less care with error handling exists. Overall, the applications chosen should be mature and widely used.

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

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

The complete list of applications is shown in Table 3.1.

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

To perform the source code analysis two parsers were generated using antlr [Parr2006], for C#, and javacc [Javacc2008] for Java. These parsers were then modified to extract all the exception handling code into one text file per application. These files were then manually examined to build reports about the content of exception handlers.

We examined the source code of all applications, except for Mono. Indeed, due to its huge size, on Mono we focused on its "corlib" module.

The parsers were also used to identify and collect information about try blocks inside loops (i.e., detect try statements inside while and do..while loops). The reason why we have done this was because this type of computation can correspond to retrying a block of code, which was responsible for raising an exception, in order to recover from an abnormal situation.

The main objective of this study was to understand how programmers use the exception handling mechanisms available in programming languages. Nevertheless, the analysis of the applications source code is not enough by itself when trying to distinguish between the exceptions that the programmer wants to handle and the exceptions that might occur at run-time. The main reason for this is that the generated code (the product of the source

code compilation) can produce more and different exceptions than the ones that are declared in the applications source code by means of `throw` and `throws` statements.

Table 3.1 – Applications listed by group.

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

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

To perform the analysis of the .NET assemblies¹ and of the Java class files two different applications were developed: one for .NET and another one for Java. To develop the analysis software for .NET, we were also forced to create and use our own IL code instrumentation library because none was available for that platform at the time. Thus, we created and used the *RAIL assembly instrumentation library* [Cabral2005] to access assembly metadata and IL code and extract all the information about exceptions, exception handlers and exception protection blocks in .NET assemblies. The second application targeted the Java platform and used the *Javassist bytecode engineering library* [Chiba2000] to read class files and extract exception handlers' information.

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

For each application only one file or package of classes was analyzed. Table 3.2 shows the names of the files and packages that were used in this study. The criterion followed to select these targets was the size of the files (larger files were preferred) and their relevance in the implementation of the application core (more relevant ones were preferred).

When performing the parsing of the applications source code, both for Java and .NET applications, we only had to consider the identification of the protected regions of code (try blocks), of the handlers and finalizers for those blocks (catch and finally blocks), and the occurrence of throw statements in the code (and throws in Java), while, when

¹ A .NET Assembly is a PE (portable executable) file for Windows GUI on Intel x86. There are two kinds of these fyles: process assemblies (EXE) and library assemblies (DLL). An assembly is composed by metadata and IL (Intermediate Code). IL code is the "machine code" executed by the .NET platform runtime.

performing the analysis of the executable files, we had to look not only into the code (IL and bytecode) but also, and more importantly, to the metadata inside the files.

Table 3.2 - List of Assemblies and Java Packages analyzed.

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

In a .NET assembly all the metadata is organized into tables [ISO23271:2006]. There are tables that hold information about the types defined and referenced in the assembly, modules, methods, parameters, resources, etc. Entries in each table can reference other tables and even other entries in different tables through *tokens* (encoded index values). For instance, each entry in the *Method* table has the following fields: *RVA* (4 byte constant); *ImplFlags* (bitmask); *Flags* (bitmask); *Name* (index into *String heap*); *Signature* (index into *Blob heap*); *ParamList* (index into *Param table*). The *ParamList* attribute is a token value that represents an index into another metadata table, the *Params* table. In the same sense, *Name* is a relative pointer to the zone in memory where all the strings used in the assembly are stored and *Signature* is a memory pointer into the zone where all unsorted byte streams within the assembly are kept.

The *Method* table only contains information about the method (or pointers to the location of that information), it does not contain any actual IL code or information about the method body. The IL code for all the methods in the assembly is kept on another section of the assembly. In this section, IL methods are organized in three different parts: *header*, *body*, and *extra data sections*. Currently, these data sections are used to store information about the exception handling code in the method. A typical exception handling data section

contains several *clauses*, each clause identifies the start point of a `try` block and the block length in bytes, the start point of an handler and the handler code length in bytes, the type of the handler (*type-based* or *catch*, *finally*, *filter* or *fault*), the token for the identification of the exception type in a type-based handler, and the offset into the code for filter-based handlers. With this information and the parsing of the IL code instructions in each method body, we were able to isolate the exception handling code inside the assemblies studied.

In Java the process is similar, but `.class` files are very simple, in terms of metadata and size, when compared to `.NET` assemblies. For example, a `.class` file only implements one class while an assembly contains several modules, each composed by one or more files which contain the metadata and code of all classes in the application.

A fundamental difference between the information available about exception handlers in Java and `.NET` is that Java does not provides any data about the length (or the identification of the) final instruction of exception handlers [Gosling2005] like `.NET` does.

3.2.2. Results

In the following sections we will present the results of this study, drawing some observations about their significance. The topics under analysis were already presented at the beginning of this section, but for the sake of easiness we will enumerate them once again: (a) percentage of exception handling code; (b) the type of actions performed inside exception handlers; (c) the classes used as exception handler arguments; (d) the exception types most frequently caught; (e) the call stack levels that an exception travels before it is caught; (f) the size of handlers; (g) the types of handlers; (h) the usage of checked or unchecked exceptions; (i) the usage of a *retry*-like functionality.

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

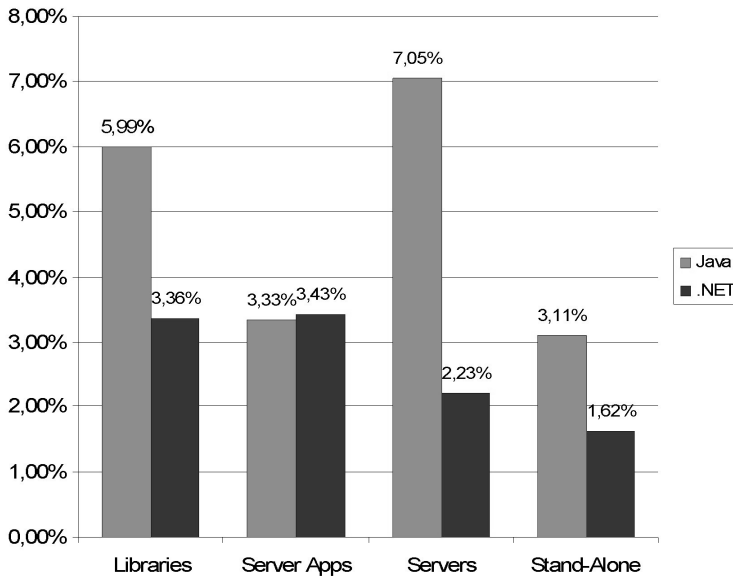


Figure 3.1 – Amount of exception handling code

Error Handling Code in Applications

One important metric for understanding current error handling practices is the percentage of source code that is used in that task. For gathering this metric, we compared the number of lines of code inside all `catch` and `finally`¹ handlers to the total number of lines of the program. The results are shown in Figure 3.1.

It is quite visible that in Java there is more code dedicated to error handling than in .NET. This difference can be explained by the fact that in Java it is compulsory to handle or declare all exceptions a method may throw, thus increasing the total amount of code used for error handling. Curiously, there is an exception to this pattern. In the Server Application group, the difference is almost non-existent. To explain this result we examined the applications' source code. For this class of applications, both in Java and .NET, programmers wrote quite similar code. Meaning that they expect the same kind of errors (e.g., database connections loss, communication problems, missing data, etc.) and

¹ Note that `finally` code blocks are not really exception handlers in the technical sense of the word. Even so, for simplicity, we will use the term “`finally` handler” when referring to code blocks related to clean-up actions. The same term is used in the ECMA specs of the CLR.

they use the same kind of treatment (the most common handler action in this type of applications is logging the error).

One surprising result is that the total amount of code dedicated to exception handling is much less than what would be expected. This is even more surprising in Java where using exceptions is almost mandatory even in small programs. Our results show that the maximum amount of code used for error handling was 7% in the Servers group. Overall, the result is 5% for Java, with a 2% standard deviation, and 3% for .NET, with a standard deviation of 1%. It should be noted that, as we have already mentioned in the prior section, the applications used in this study are quite mature, being widely used.

We reason that the effort dedicated to writing error protection mechanisms is not as high as expected, even for highly critical applications like servers. The forceful of declaring and catching checked exceptions in Java effectively increases (almost doubles) the amount of error handling code written, even though it is still represents a small fraction of all the code of an application. The critical issue is that normally error handling code is being used more to alert the user, to abort the applications or to force them to continue their execution, than to actually recover from existing errors.

The amount of exception handling code in the Stand-Alone group is smaller than the amount of error handling code in infrastructure software. This is unexpected if we consider exception handling to be an application-specific error recovery technique. A simple explanation can be that infrastructure software needs to run 24x7 with a minimum of human supervision. This fact may influence developers to produce “better” (albeit more complex) recovery code, whereas stand-alone applications developers can limit their error handling code to warnings, hoping that the user is able to correct the cause of the erroneous behavior. In the next sections we will provide an in depth analysis of the error handling actions present in the four application groups.

Code in Exception Handlers

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

To be able to report on this subject we had to inspect sets of ten thousand lines of application source code. As a matter of fact, we covered all the handlers (`catch` and `finally`) in all the applications except for *JBoss* and *Eclipse*. For these two, due to their

dimension, only 10% of the 96 405 lines of code existing inside of exception handlers were examined. Even so, we believe that they are representative of the rest.

To simplify the classification of these error handling actions, we propose a small set of categories that enable the grouping of related actions. These categories are summarized in Table 3.3.

Table 3.3 - Description of the Handler's actions categories.

Category	Description
Empty	The handler is empty. It has no code and does nothing more than cleaning the stack.
Log	Some kind of error logging or user notification is carried out.
Alternative Configuration	In the event of an error or in the execution of a finally block some kind of pre-determined (alternative) object state configuration is used.
Throw	A new object is created and thrown or the existing exception is re-thrown.
Continue	The protected block is inside a loop and the handler forces it to abandon the current iteration and start a new one.
Return	The handler forces the method in execution to return or the application to exit. If the handler is inside a loop, a break action is also assumed to belong to this category.
Rollback	The handler performs a rollback of the modifications performed inside the protected block or resets the state of all/some objects (e.g., recreating a database connection).
Close	The code ensures that an open connection or data stream is closed. Another action that belongs to this category is the release of a lock over some resource.
Assert	The handler performs some kind of assert operation. This category is separated because it happens quite a lot. Note that in many cases, when the assertion is not successful, this results in a new exception being thrown possibly terminating the application.
Delegates (only for .NET)	A new delegate is added. The delegate object contains information about what to do when a specific event occurs.
Others	Any kind of action that does not correspond to the previous ones.

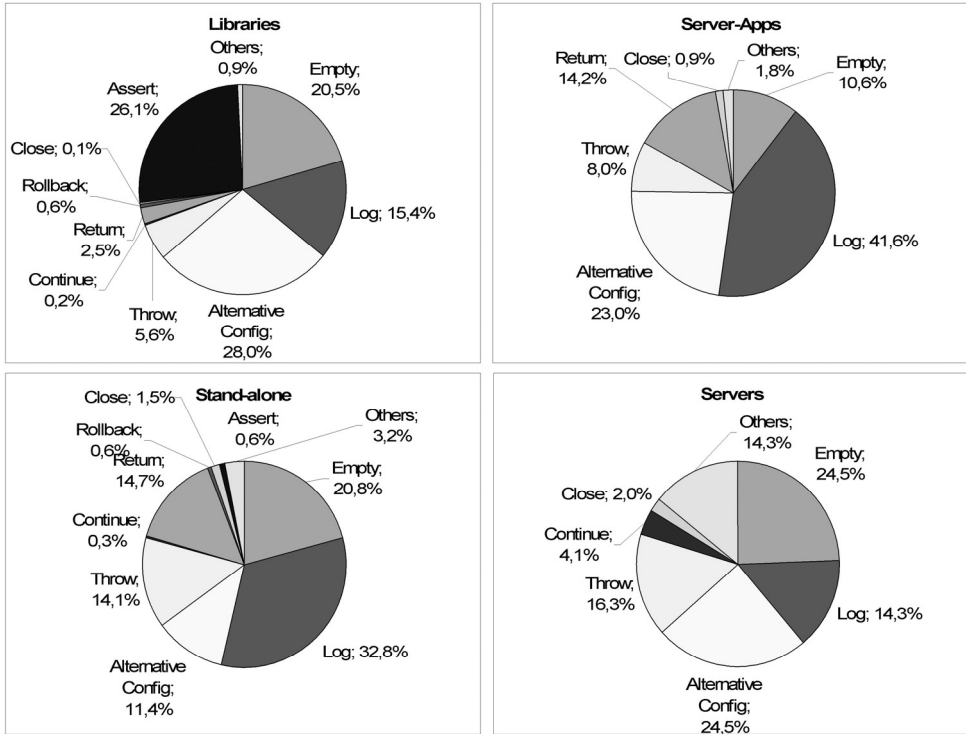


Figure 3.2 – Catch handler actions for .NET programs.

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

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

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

The graph of Figure 3.2 shows the average of results by application group for .NET catch handlers. In the four application groups 60% to 75% of the total distribution of handler actions is composed of three categories: Empty, Log and Alternative Configuration.

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

Logging errors is also one of the most common actions in the handlers of all the applications. In fact, it is the most common action in Server-Apps and Stand-alone groups. Considering web applications and desktop applications, this typically corresponds to the generation of an error log, the notification of the user about the occurrence of a problem and the abortion of the task. This idea is re-enforced by the value of the Return action category in these two application groups which is the identical and the highest of all four groups.

The number of Alternative Configuration actions reports on the usage of alternative computation or object’s state reconstruction when the code inside a protected block fails in achieving its objective. These actions are by far the most individualized and specialized of all. In some cases they are used to completely replace the code inside the protected block.

In the Libraries applications group, Assert operations are the second most common error handling action. Asserts ensure that if an error occurs, the cause of the error is well known and reported to the user/programmer. In Servers there is also a high distribution value for the Others category. These actions are mainly related with thread stopping.

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

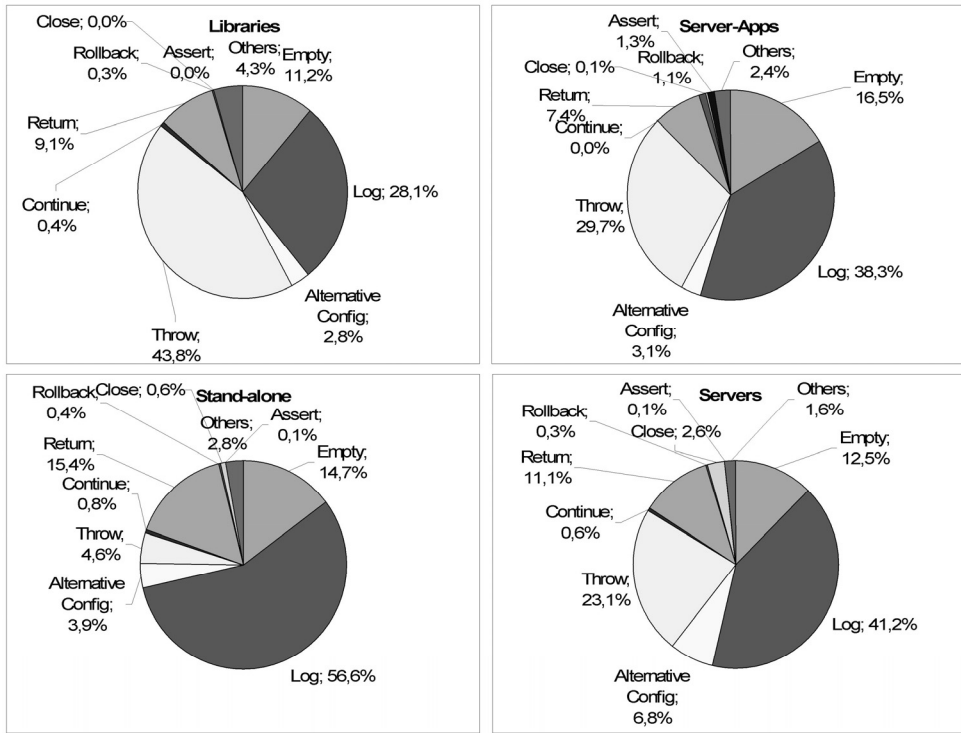


Figure 3.3 - Catch handler actions for Java programs.

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

Figure 3.3 shows the results for catch handlers in Java programs. Only in the Stand-alone and Server-Apps groups we found some similarity with .NET. Despite this fact, it is possible to see the same type of clustering found in .NET. The cluster of categories that concentrate the highest distribution of values is composed by Empty, Log, Alternative Configuration, Throw and Continue actions.

The distribution values on the Empty category surprised us once again. This value is lower than the ones found in .NET. This suggests that the checked exception mechanism has little or no weight on the decision of the programmer to leave an exception handler empty: another reason must exist to justify the existence of empty handlers besides silencing exceptions. In .NET this happen quite frequently for building alternative execution blocks. We risk saying that in Java exception mechanisms are no longer being used only to handle

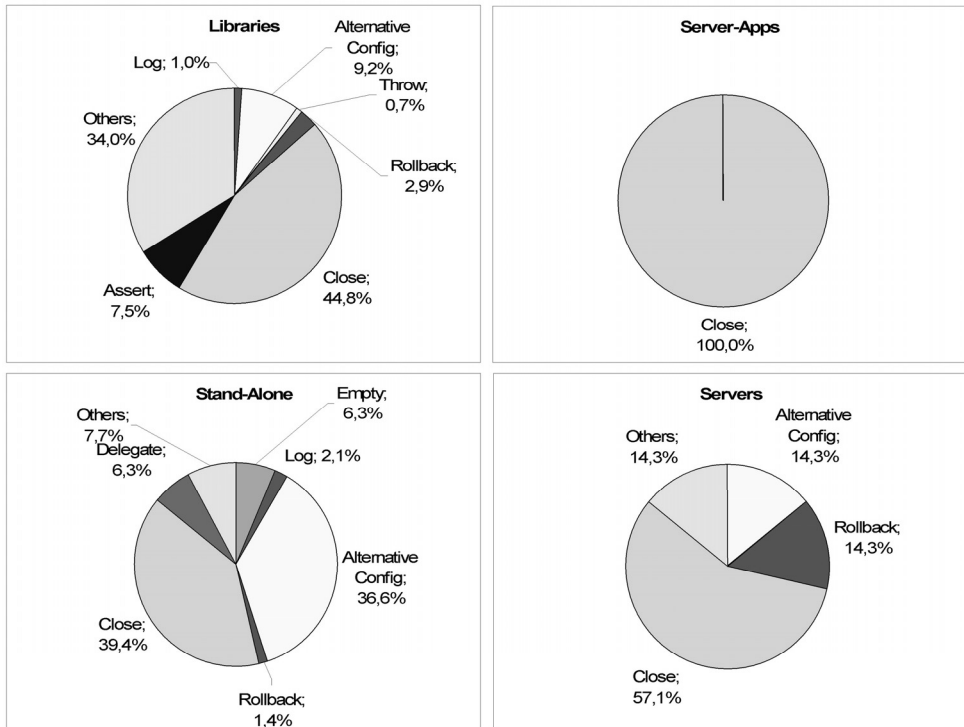


Figure 3.4 – Count of actions for **Finally** handlers in .NET programs.

“exceptional situations” but also as control/execution flow construct of the language. (Note that even the Java API sometimes forces this. For instance, the detection of an end-of-file can only be done by catching an exception.)

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

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

Figure 3.4 illustrates the results for **finally** handlers in .NET. The distribution of the several actions is different from the one found in catch handlers. Nevertheless, it is visible that the most common handler action category in .NET, for all application groups, is Close.

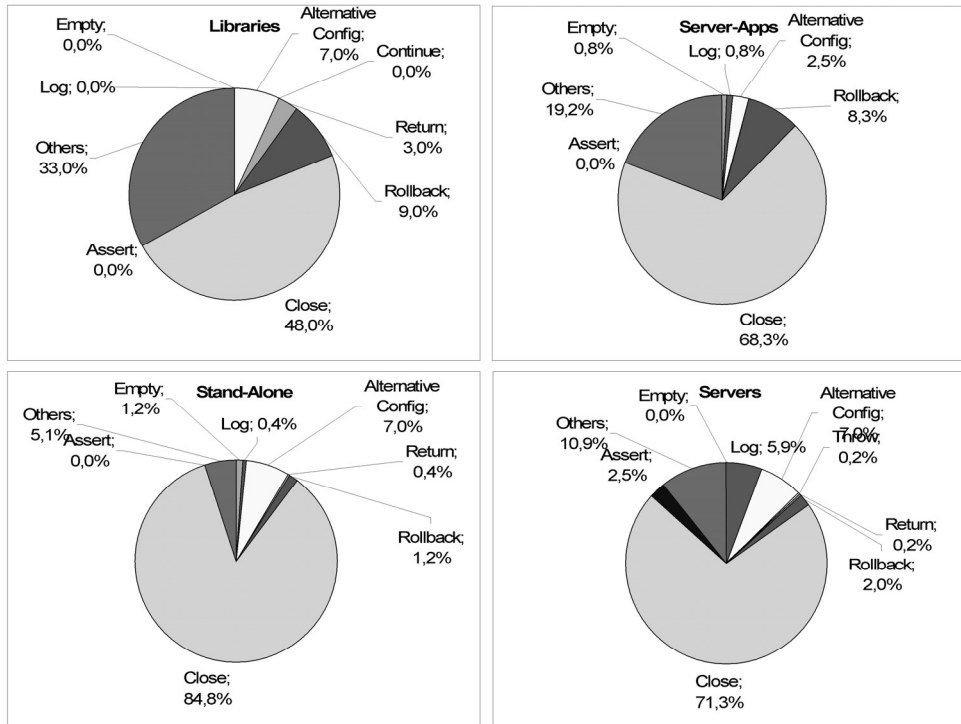


Figure 3.5 - Count of actions for **Finally** handlers in Java programs.

I.e., finally handlers, in our test suite, are mainly used to close connections and release resources.

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

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

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

In Java applications (Figure 3.5) the scenario is very similar to the one found in .NET. Close is the most significant category in all application groups. There are also some actions classified as Others, which are similar to the ones of .NET. In Java they have more weight in the distribution, indicating a higher programming heterogeneity in exception handling.

Rollback and Alternative Configuration actions are also used as handler actions in Java `finally` handlers.

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

Classes Used as Exception Handler Arguments

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

Figure 3.6 shows the most common classes used as argument of catch instructions in .NET applications. The results are grouped by application type and the values represent the weighted average of the distribution among applications of the same group. Thus, programs with the largest number of handlers have more weight in the final result.

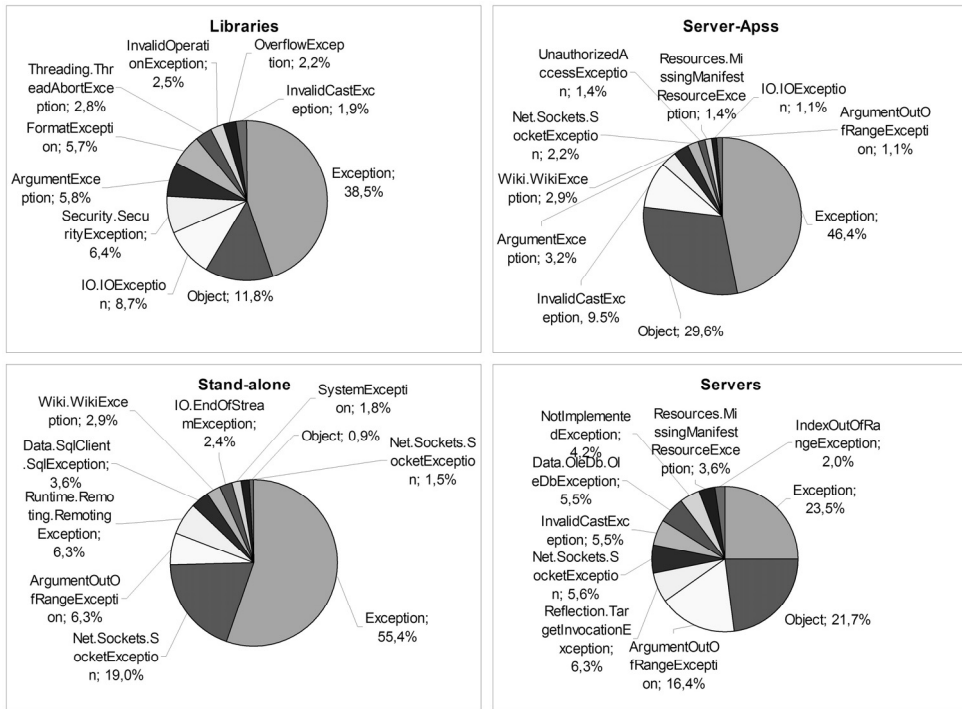


Figure 3.6 – .NET classes being used as catch arguments.

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

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

I/O related exception handlers are fairly used in Libraries and Servers. Also invalid arguments types, number and format errors are treated as exceptions by all the

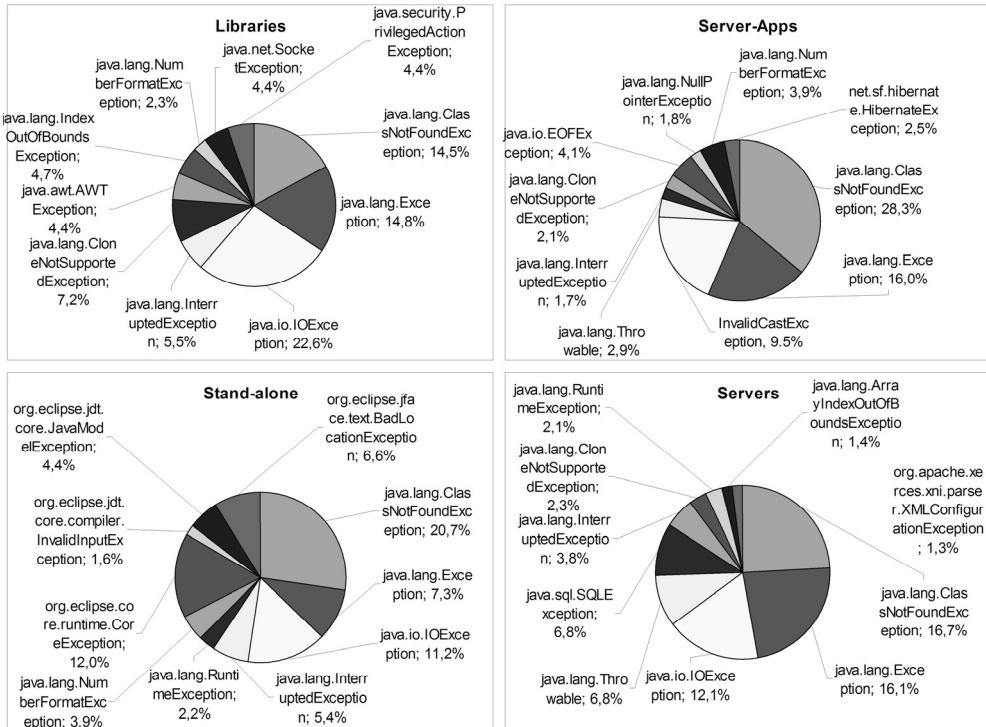


Figure 3.7 – Java classes being used as catch arguments.

applications as shown by the presence of `System.ArgumentException` handlers and `System.FormatException` handlers.

There are not many differences between Java and .NET in terms of catch arguments. Figure 3.7 shows the results for Java. It is possible to conclude that the most generic exception classes are the preferred ones: `Exception`, `IOException`, and `ClassNotFoundException`. We tried to find out why `ClassNotFoundException` is so commonly used by analyzing the source code. For the most part, most of the handlers associated to the use of this class are empty, just log the error or throw a new kind of exception. Others try to load a parent class of the class not found or another completely different class. In general, these handlers are associated with “plug-in” mechanisms or modular software components using dynamic class loading. An example is the way JDBC database drivers are loaded by using `Class.forName()`. [Gosling2005]

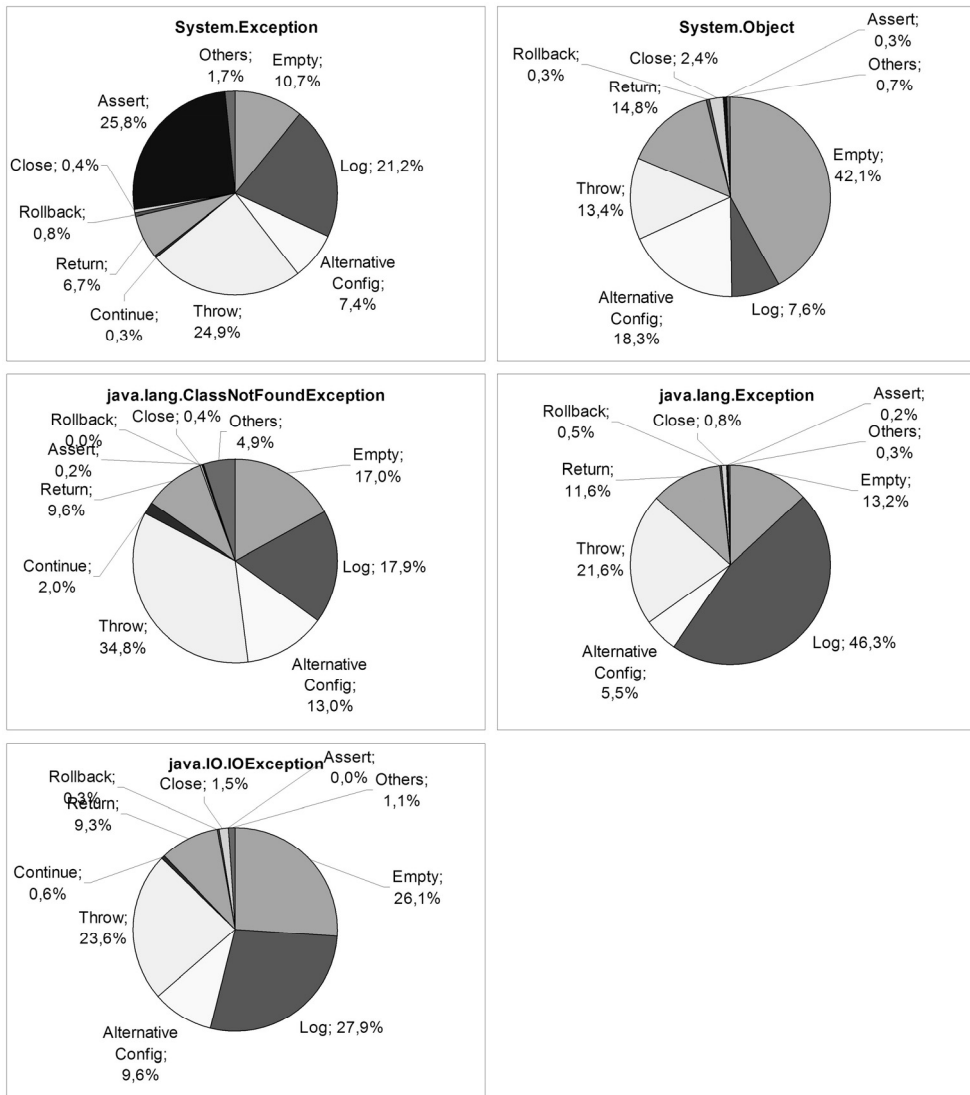


Figure 3.8 – Handler actions distribution for the most used catch handler classes.

Finally, we did an analysis of all the applications source code to find out what was the distribution of handler actions by catch handler argument class for the most commonly used classes. The results can be found in Figure 3.8 and quite different from one type of exception class to another. Even so, it is still possible to say that the dominant handler actions are the ones belonging to the categories: Empty, Log, Alternative Configuration, Throw and Return.

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

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

Handled Exceptions

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

Exception Universe

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

- All the exceptions that the bytecode instructions of a method may raise accordingly to the Java specs [Gosling2005];
- All the exception classes declared in the `throws` statement of the methods being called;
- All the exceptions that are produced inside a protected block and are caught by one of its handlers;
- All the exception classes in the method own `throws` statement.

In .NET, finding the exceptions that might be thrown is a more difficult task. Indeed, because there are no checked exceptions, to discover what exceptions a method may raise it is necessary to know:

- All the exceptions that can be raised by each one of the IL instructions accordingly to the ECMA specs of the CLR [ISO23271:2006];
- All the exceptions that the method being called may raise;
- All the exception classes present in explicit `throw` statements;
- All the exceptions that are produced inside a protected block and are not caught by one of its handlers.

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

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

To obtain meaningfully results we decided to perform a second analysis not using all the data from the static analysis of bytecode and IL code instructions. In particular, we filtered out a group of exceptions that are not normally related to the program logic, and that the programmer should not normally handle, considering the rest. The list of exceptions that were filtered out is shown in Table 3.4.

Table 3.4 - Java and .NET exception classes for bytecode and IL code instructions.

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

Results for handled exceptions

Being aware of the complete list of exceptions that an application can raise and of the complete list of handlers and protected blocks, it is possible to find out which ones are the most commonly handled exception types. The results for .NET applications are shown in Figure 3.9. The values represent the average of results by application group where every application had a different weight in the overall result according to the total number of results that they provided. It is possible to observe that the results are very different between application groups. For instance, in the Libraries group, the most commonly handled exceptions are `ArgumentNullException` and `ArgumentException`, resulting from bad parameter use in method invocations. In the remaining three groups the number one exception type is `Exception`, this can be a symptom of the existence of a larger and more differentiated set of exceptions that can occur. If many different exceptions can occur it is viable to assume that the most generalized type (i.e., `Exception`, `IOException`, etc.) becomes the most common one.

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

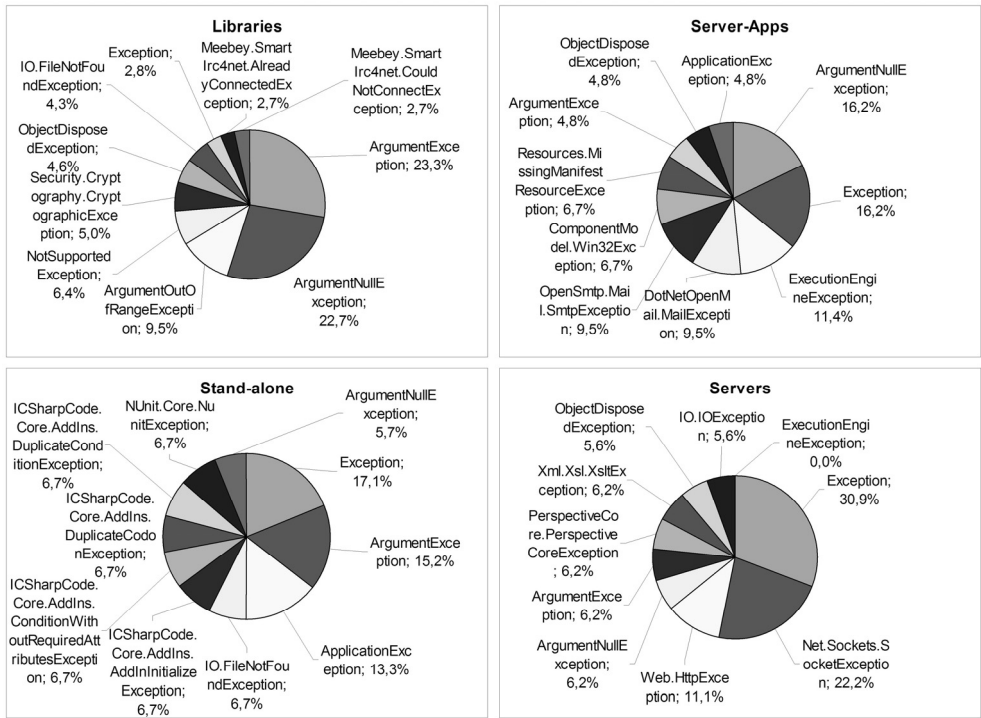


Figure 3.9 – Most commonly handled exception types in .NET.

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

In both Java and .NET, there is a large spectrum of exception types being handled. The results for Java are illustrated in Figure 3.10. IOException is the most “caught” exception type in all application groups. It is also possible to observe that the exception types are tightly related to the applications. For instance in Stand-alone applications, three of the exception classes are from Eclipse. Due to its huge size Eclipse carries a large weight in its application group results and, as we are able to observe, its “private” exceptions are present in this top ten.

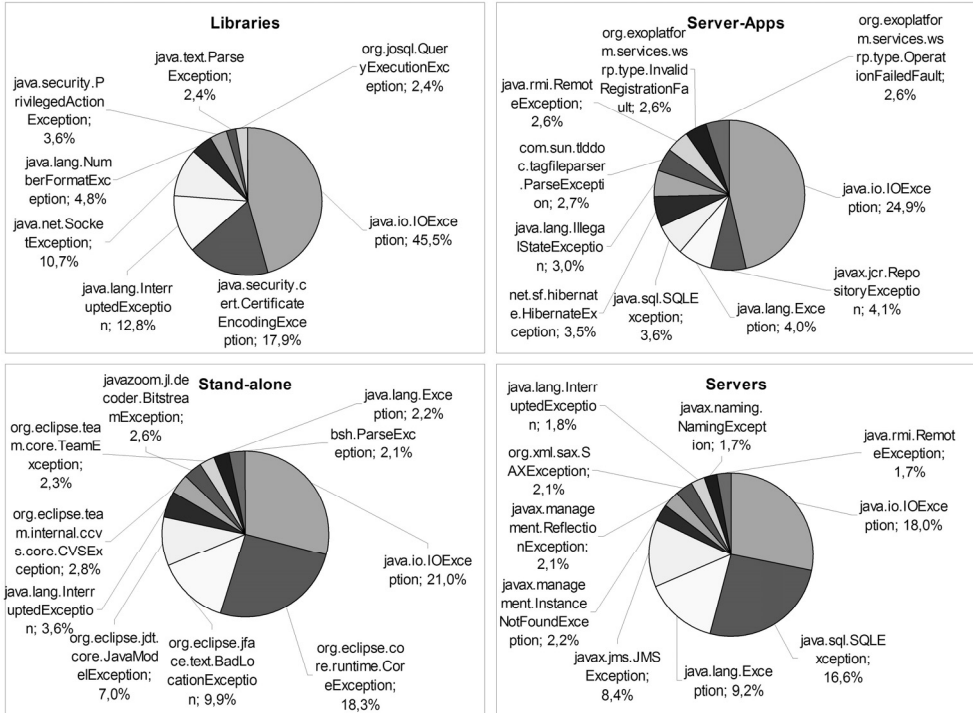


Figure 3.10 – Most commonly handled exception types in Java.

Call Stack Levels Analysis

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

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

In Figure 3.11 it is possible to observe that in Java almost 80% of the exceptions are caught one level up from where they are generated, 15% two levels up, 5% three levels up and all the remaining are caught as high as five levels. On the other hand, in .NET, exceptions can cover up to seventeen levels and the distribution of the exceptions per levels covered is much sparser than in Java. The .NET programmer is not forced to catch exceptions and, as

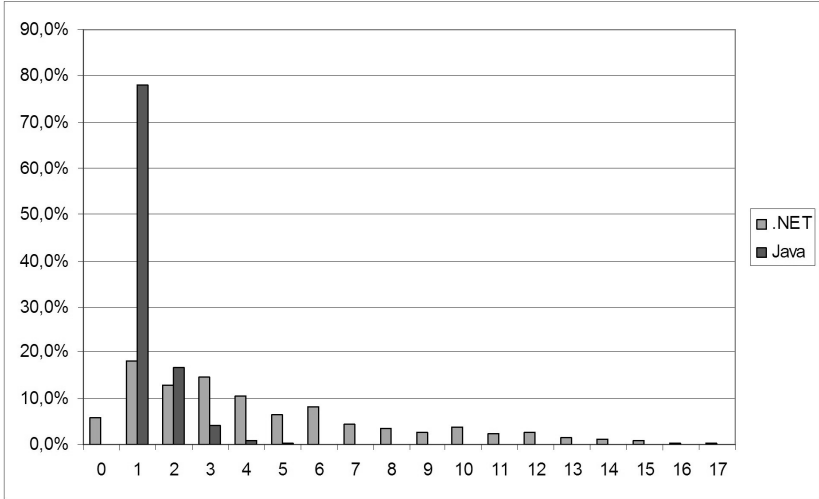


Figure 3.11 – Call stack levels for caught exceptions.

a result, exceptions can be caught much later in the call stack and most times by exception handlers with general catch arguments.

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

Handler size

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

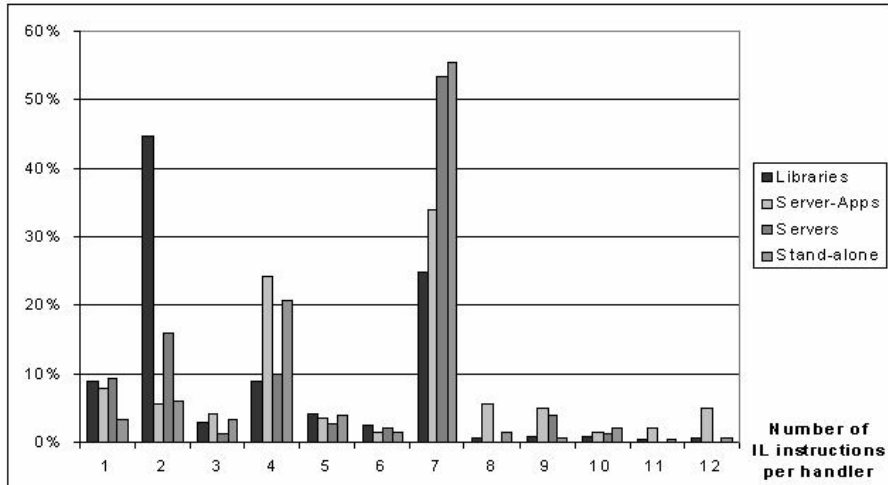


Figure 3.12 – Handlers size in number of IL code instructions for .NET.

The graph in Figure 3.12 shows that the largest set of handlers in Server-Apps, Servers and Stand-alone applications groups have 8 IL Code instructions. In the Libraries group more than 40% of the handlers have 3 instructions. The second largest set of handlers in all groups has 5 instructions. Obviously, there are larger handlers but they are so scarce that we excluded them from the graph to improve its readability.

To understand the dominance and content of the small handlers, we analyzed the full IL code in all handlers. We found the following interesting facts:

- In the 526 handlers with size 8, 500 (95%) invoked a `Dispose()` method in some object; from this 500 there were two major sets of handlers with the exact same opcodes, one with 329 elements and the other with 166; the remaining 5 handlers were different between them; these handlers were all `finally` handlers;
- In the set of handlers with 5 instructions there were 194 elements; 74 disposed of some object; 24 created and threw a new exception; 36 stored some value;
- 484 of the 498 handlers of size 3 were `finally` handlers; 426 handlers had exactly the same opcodes and were responsible for closing a database connection; other 34 handlers also had the same code and invoked a `Finalize()` method in some object;

- The largest set of handlers with size 2 was empty handlers in the source code and its actions consisted in cleaning the stack and returning; others re-threw the exception, and the rest called some `Assert()` method.

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

Types of handlers

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

The data in Table 3.5 shows that for the 1565 protected blocks found in the .NET applications there are 1630 handlers; 1144 protected blocks (73%) have `finally` handlers; but only 29% have catch handlers. On Java there are 18389 handlers distributed by 17024 protected blocks; 8109 protected blocks (48%) have `finally` handlers; 9402 (55%) have catch handlers.

Table 3.5 – Number of protected blocks, catch handlers and finally handlers.

	Protected Blocks	Handlers	Protected Blocks with Finally Handlers	Protected Blocks with Catch Handlers
.NET	1565	1630	1144 (73%)	450 (29%)
Java	17024	18389	8109 (48%)	9402 (55%)

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

In the graph of Figure 3.13 it is possible to see that Java applications have higher maximum values of catch handlers per protected block, while the average number of catch blocks per try block is almost identical in all the application groups for the two platforms and has the approximate value of one. The standard deviation values are also very low meaning that the largest number of protected blocks has only one catch handler.

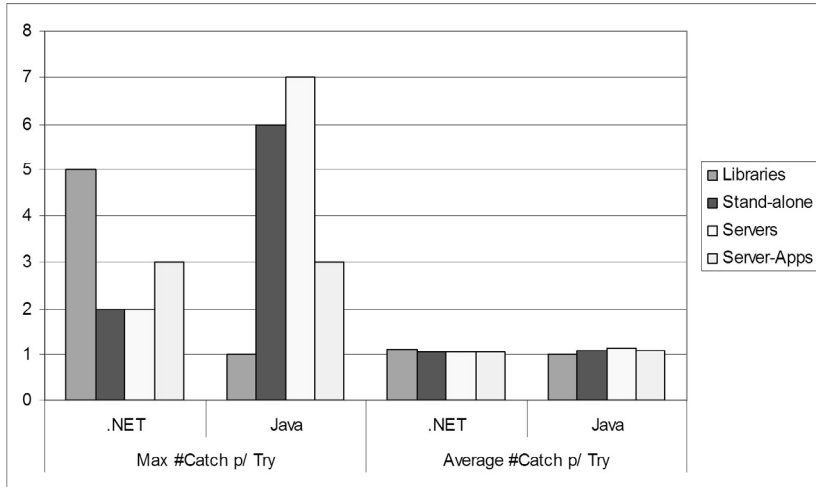


Figure 3.13 - Number of catch handlers per protected block.

Checked vs. Unchecked Exceptions

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

We compared the number of catch instructions that have an unchecked exception class as argument with the total number of catch instructions. The results are shown in Table 3.6. It is possible to observe that except for the Stand-Alone application group, where the usage reaches 36.7%, for the remaining groups, values are very low, never exceeding 9%. Nevertheless, unchecked exceptions are indeed being used and, besides their extensive usage by some dedicated libraries, they are largely used to report on underlying system errors.

Table 3.6 - Usage of Unchecked exceptions in Java catch handlers.

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

Retry functionality

Neither Java or .NET have nothing like a “retry” block functionality that would enable the programmer to execute a try block in a loop until it succeeds or reaches a certain condition. Other languages like Smalltalk [Goldberg1989] or Eiffel [Meyer1988] have this kind of construct.

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

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

This can be considered a sort of blind analysis, since we do not know if the programmer really intended to do a “retry”. Nevertheless, 6% of all catch handlers in both Java and .NET were inside loops and if the programmer really intended to do a “retry”, which appears to be the most reasonable reason, that would be a fairly interesting result to justify the addition of this functionality to both languages.

3.2.3. Related work

Since the pioneering work of John B. Goodenough in the definition of a notation for exception handling [Goodenough1975] and Flaviu Cristian in defining its usage [Cristian1980], several studies¹ have been conducted over the years for validating the options taken in each different implementation.

For instance, Alessandro Garcia, et al. did a comparative study on exception handling mechanisms available developing dependable software [Garcia2001]. Garcia’s et al. work consisted in a survey of exception handling approaches in twelve object-oriented languages. Each programming language was analyzed in respect to ten technical aspects associated with exception handling constructs: (1) exception representation; (2) external exceptions in signatures; (3) separation between internal and external exceptions; (4) attachment of handlers to program constructs (e.g., to statements, objects, methods, etc.);

¹ Some of these studies have already been discussed in greater detail in Chapter 2 (when we assessed the state of the art in exception handling). Nonetheless, for providing a suitable related work for this section it is important to refer them here once again. We will only include brief descriptions and literature references for the subjects that have already been addressed.

(5) dynamism of handler binding; (6) propagation of exceptions; (7) continuation of the flow control (resumption or termination); (8) clean-up actions; (9) reliability checks; (10) and concurrent exception handling. After the evaluation of all the programming languages in terms of exception mechanisms, the major conclusion of the study was that “none of the existing exception mechanisms has so far followed appropriate design criteria” and programming language designers are not paying enough attention to properly supporting error handling in programming languages.

Saurabh Sinha and Mary Jean Harrold performed an extensive analysis of programs with exception handling constructs and discussed their effects on analysis techniques such as control flow, data flow, and control dependence [Sinha2000]. In the analysis, the authors also presented techniques to create intraprocedural and interprocedural representations of Java programs that contain exception handling constructs and an algorithm for computing control dependences in their presence. Using that work, the authors performed several studies and showed that 8.1% of the methods analyzed used some kind of exception mechanism and that these constructs had an important influence in control-dependence analysis.

R. Miller and A. Tripathi identified several problems in exception handling mechanisms for Object-Oriented software development [Miller1997]. In their work, it is shown that the requirements of exception handling often conflict with some of the goals of object-oriented designs, such as supporting design evolution, functional specialization, and abstraction for implementation transparency. Being specific: object-oriented programming does not support a complete exception specification (extra information may be needed for the exception context not supported by an object interface); state transitions are not always atomic in exception handling; exception information needs to be specific, but functions can be overloaded to have a different meaning in different situations; the exception handling control flow path is different from the normal execution path and is up to the programmer to differentiate both of them. Thus, modifying an object-oriented framework to incorporate an exception handling mechanism can have a negative impact. In the worst case we can expect the introduction of partial states into the abstraction, the loss of object encapsulation due to internal exception information leaking, a decrease in modularity, and inheritance anomalies.

Martin P. Robillard and Gail C. Murphy in their article on how to design “robust Java programs with exceptions”, classified exceptions as a global design problem and discussed the complexity of exception structures [Robillard2000]. In their work, the authors pointed

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

More recently, due to a new Aspect Oriented Programming (AOP) approach to exception handling, two interesting studies were published emphasizing the separation of concerns in error handling code writing [Lippert2000;Filho2005]. Martin Lippert and Cristina Lopes rewrote a Java application using AspectJ. Their objective was to provide a clear separation between the development of business code and exception handling code. This was achieved by applying error handling code as an advice (in AOP terminology) [Elrad2001]. With this approach they also obtained a large reduction in the amount of exception handling code present in the application. Their results show that without aspects, the amount of code for exceptions is almost 11% of all the code; with aspects it represents only 2.9%. Lippert's paper also accounts the total number of catch blocks in the code and the most common exception classes used as parameters for these catch statements. One of the measures they present to support their AOP approach is the reduction of the number of different handlers effectively written for each one of the most commonly used exception classes. For the top 5 classes it was observed a reduction in the number of implemented handlers between 90.0% and 96.5%. F. Filho and C. Rubira conducted a similar study but they were not so enthusiastic in their results. The authors presented four metrics to evaluate the AOP approach to exception handling: separation of concerns; coupling between components and depth of inheritance tree; cohesion in the access to fields by pairs of method and advice; and dimension (size and number) of code. The work reports that the improvements of using AOP do not represent a substantial gain in any of the presented metrics showing that reusing handlers is much more difficult than is usually advertised. Handler reuse depends of the type of exceptions being handled, on what the handler does, the amount of contextual information needed, and what the method raising the exception returns and what the throws clause actually specifies.

The objective of our study is different from its predecessors. It does not directly target the quality of the mechanisms available in programming languages but the usage that programmers make of them. The emphasis is on understanding how programmers write

exception handling code, how much of the code of an application is dedicated to error recovery and identifying possible flaws in their usage.

Recently, Hina Shah et al. have made the question “Why Do Developers Neglect Exception Handling?” [Shah2008a]. In their paper, the authors explore the problems associated with exception handling from a new dimension: *the human*. The article describes a study where the focus was on evaluating different perspectives of software developers to understand how they perceive exception handling and what methods they adopt to deal with exception handling constructs. The authors also mention that, based on previous studies, they have developed a tool for visualizing the exception handling constructs inside programs. In this study, the usefulness of such tool, as a software development aid, was assessed.

Based on the results from a previous survey¹, Shah et al. have created the ENHANCE [Shah2008b] software (ExceptioN HANdling Centric visualization). ENHANCE offers three views of the exception handling constructs inside Java programs:

- The *Quantitative View* presents high-level information about throw-catch pairs at the level of packages, classes, or methods;
- The *Flow View* provides details about multiple exceptions flows at abstracted level; the view presents type definitions, throw clauses, and catch clauses as abstract icons on separate layers; exception flow is represented as links between them;
- The *Contextual View* uses an abstract code view of the system where exception-handling constructs and their flows are put into perspective.

The preliminary evaluation of the ENHANCE tool involved three graduate students from the software-engineering group at the Georgia Institute of Technology. The results were surprising - “the participants often ignored exception-handling constructs. (e.g., the *ignore-for-now approach* in which developers ignore exception handling until there is an error or until they are forced to address it).” [Shah2008a]

In order to better understand the significance of the previous results, the authors extended their study to include insights from industry software developers¹. The study focused on

¹ Survey conducted with 34 software developers to understand their needs in terms of exception-handling constructs in Java programs.

understanding the approach software developers adopt to deal with exception handling-related tasks, such as designing, coding, reviewing, refactoring, testing, and debugging.

These results were consistent with the ones of the initial study. To avoid giving a misguided interpretation of the study participant answers, we will be quoting Shah et al. on their own explanation of the interviewees' answers:

- “All the participants we interviewed stated that they use exception handling primarily for debugging purposes”...“they use the names of the exceptions to understand the context of the surrounding program code. However, most of the participants agreed that in cases where Java’s defined exceptions (e.g., `ClassNotFoundException`) are used, they tend to ignore understanding the exception handling implemented around these exceptions.”
- “The only exception to the *ignore-for-now* behavior occurs in scenarios where the code on which the participants were working already had some useful implementation of exception handling. In such scenarios, the participants agreed that they try to mimic the existing code. Thus, in general, participants try to avoid handling exceptions unless some support structure is already available.”
- “They [the participants] explained this by stating that they will not use exception handling if the compiler does not prompt them with compile-time errors when the appropriate exception-related code was missing (e.g., declaration of `throws` clause, or implementation of respective `try-catch` block). Therefore, the two similar attitudes of “avoiding exception handling” that developers carry on both the occasions—when no support for exception handling is available and when support is available—indicate that developers are less willing to deal with exception handling.”
- “Another common attitude held by the participants is that they do not think that exception handling is a high-priority task. Participants think that it is time consuming and hence, a waste of time, to design exception-handling code in advance.”

From this analysis we can conclude that there has been an evolution on the way exception handling is used. Originally, exception handling mechanisms were designed to do error

¹ Eight developers with three to ten years of experience plus one with more than ten years.

handling and recovery (proactively) but, nowadays, developers use them as a debugging tool (reactively) and a way to understand programs. Additionally, it is safe by now to say that developers “tend not to invest time in implementing code for proper handling of error conditions unless its implementation helps with debugging”.

As a consequence of such approach to exception handling, error handling code exhibits poor quality. And, forcing developers to implement exception handling code is not a solution - when such policy is in use, developers tend not to implement code as thoughtfully as it would be necessary.

Our study and Shah’s study agree on two fundamental aspects - developers are not satisfied with the existing exception handling mechanisms; and the complexity of some mechanisms is, in many cases, completely overwhelming and un-productive.

3.3. Documenting exceptions

Documentation plays a very important role in software development. This role is even more important when developing for platforms, like the .NET platform, that do not support the checked exceptions model in programming languages such as C#.

In this section, we present a study on the efficiency of existent exceptions documentation (for .NET applications) in alerting developers to the dangers involved in a method invocation. We want to know how well documentation performs such task when compared with the usage of checked exceptions. In broader terms, we determine which exceptions a selected set of programs might raise and verify which of these are documented and which are not.

3.3.1. Motivation

When all exceptions are unchecked, programmers are not forced to declare a method as thrower of an exception, and so, the relation between a method and the exceptions it can throw is weaker. Thus, in the absence of such declarations, a programmer will never be warned by the compiler if he or she forgets to handle an exception. Furthermore, a programmer using *Reflection* to access a method is not able to discover which exceptions that method throws just by looking at its declaration. Thus, .NET reflection does not give programmers access to the complete exception information of a method. Yet, it is possible

Exceptions

Exception Type	Condition
DivideByZeroException	This exception thrown when <code>parm2 = 0</code>

Figure 3.14 –Automatic documentation of an exception using specialized tags.

to utilize reflection to analyze the `throw` instructions in the code, used by programmers to throw exceptions in their code.

Of course, C# designers did not neglect the fact that programmers need to be aware of a method's behavior in certain exceptional circumstances. Their answer resides in special documentation tags that programmers can use to document their code with respect to exceptions.

Specially designed tools can then parse the code looking for those tags and automatically generate suitable documentation files (e.g., NDoc - <http://ndoc.sourceforge.net/>). In .NET, custom-formatted XML files are generated by Visual Studio .NET (VS.NET) [Microsoft2008]. A number of other tools can then convert from this XML to Compiled HTML (CHM) format and from this to HTML. For instance, writing the following tag "`<exception cref="System.DivideByZeroException"> This exception thrown when <c>parm2 = 0</c> </exception>`" before a method declaration in a C# program will result on the documentation info shown in Figure 3.14 after the execution of a automatic documentation generation tool. In Java, the same kind of tags is available and HTML files can be generated by Javadoc [Sun2004].

The exception handling code existent in C# and Java applications is available after source code compilation. At Java's Bytecode and .NET's IL code level, the information required by the exception handling mechanisms is kept in tables, being part of the class file or assembly metadata. These table entries identify: the *start* and *end* instructions of the protected blocks of code inside each method; the presence of handlers for the previously mentioned blocks; the type of exception being handled by each handler; the *start* instruction of each handler in Java; and the *start* and *end* instruction of each handler in .NET. Furthermore, in Java, it is possible to know which exceptions a method throws just by looking at the method's metadata. In .NET, for doing so, it is necessary to perform a detailed static analysis of all methods invoked, starting from the target method.

There are more differences between the two languages approaches at low level. The .NET exception model has more functionality available than the one made available at the C#

language level. As an example, .NET's metadata structures allow compilers to generate code for *Filter* handlers. These handlers allow the execution of code if an associated conditional expression evaluates to *true*.

The question that arises from the previous considerations in the scope of this section is that of the possibility to determine, by looking at exception documentation quality, the effectiveness of the unchecked exceptions approach - *is existent documentation (for .NET applications) as efficient as checked exceptions in alerting developers to the dangers involved on a method invocation?*

On the following sections we will focus on studying the existent documentation available for .NET applications and we will leave the Java platform aside for now. The reason why we are doing this is because on the .NET platform, and contrary to what happens in Java, only the unchecked exceptions model is available, making the importance and need for good exception documentation higher on .NET.

3.3.2. Methodology and Tools

The strategy followed in this work was to take a set of software components and examine both the binary file containing their code looking for unhandled exceptions, and the corresponding documentation to evaluate the extent to which one corresponds to the other.

For performing the current analysis, it was required going through every instruction in each software component. Since no access to high-level source code could be assumed (important for the analysis of commercial-off-the-shelf components), this needed to happen at a lower level. All .NET programs, regardless of the original language they are written in are transformed into the low-level common form known as Microsoft Intermediate Language (MSIL or IL), an assembly-like language which is our real object of analysis. The Runtime Assembly Instrumentation Library (RAIL) [Cabral2005] provides us with this kind of access, effectively establishing a bridge between .NET reflection, which goes as far as the method level, and IL code.

Once the tool was ready and hence a mechanism to compare code and documentation was available, a set of software components to analyze was chosen. Some of these components are core parts of certain applications while others extend the functionality of bigger infrastructures. Some were not built with re-use in mind while others were built especially

for re-use. Finally, the tool was run for each component and its documentation, and results were gathered.

The *Analyzer* tool was especially developed for conducting the tests in this work. It consists in a command-line program written entirely in C#, one of .NET's high-level languages. As input, the Analyzer receives a .NET Assembly (a DLL or EXE file) location and optionally a documentation file location (Visual Studio .NET generated XML format). A number of switches can be used to specify different options. As output, an XML report is generated. The format of this report depends on the command-line options but, in general, consists of a list of the methods found in the assembly given as input. For each of those methods, a number of exception detections are depicted. Each of those exception detections represents one of two things:

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

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

Suspects are important for two reasons. First, the Analyzer is very thorough in its analysis and although it detects huge amounts of uncaught exceptions, only a relatively small number can be realistically expected to be documented by a programmer. Second, there are situations where there is a total absence of documentation. So, suspects are a way of filtering the relevant detections.

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

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

Figure 3.15 – Dictionary: IL instruction/opcode/list of exceptions.

The code analysis process is much more complicated. It consists of going through an Assembly's members, IL instruction by IL instruction, keeping track of entries/exits into/out of try-catch blocks through the use of data structures such as stacks and queues. For each instruction in the code, four different types of detection are performed, searching for possible exceptions thrown by that instruction. The first type of detection consists of a simple search in a manually generated dictionary which associates IL instructions with the exceptions they can throw, as defined in the .NET platform specification [ISO23271:2006]. Figure 3.15 shows an extract of that dictionary. We will call this type of detection, *IL instruction detection* (ILI).

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

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

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

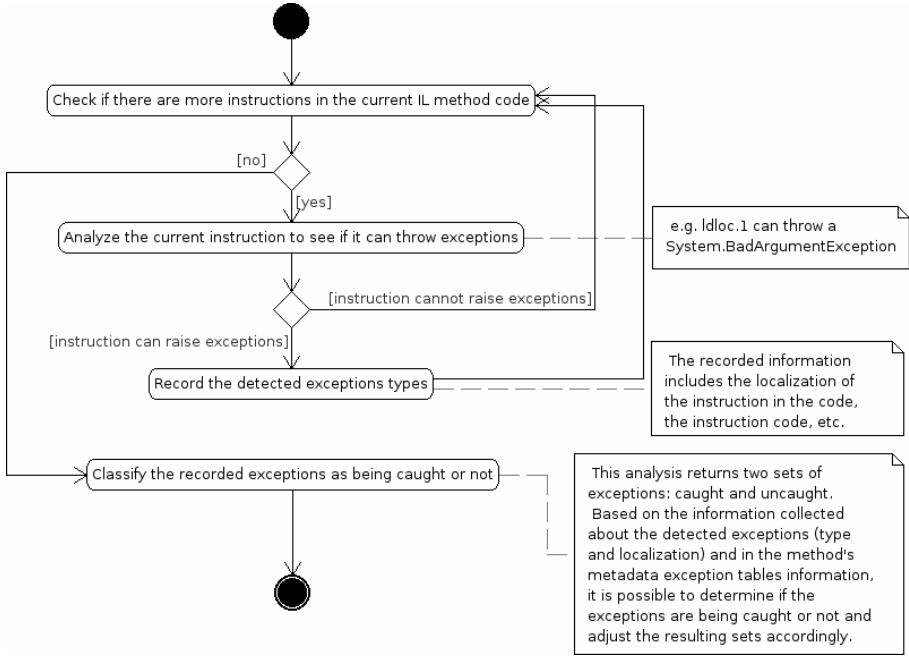


Figure 3.16 – Scheme of the code analysis process.

The code analysis process is sketched in Figure 3.16. Both caught and uncaught exceptions are represented in the final report, because it is of interest to check which types of exception programmers catch and which they do not.

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

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

The first reason is due to of the lack of penetration of the .NET platform at the time. In fact, the .NET platform was still an alternative, rather than a first choice, for developers and decision-makers. For this work, this meant having to search in forums aimed at the sharing of .NET applications (e.g., [CodeProj2008]), opposed to the more usual, like [SourceFrg2008].

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

Although the first cause is perfectly possible, especially in cases where VS.NET is not used at all (e.g., the Mono [Mono2008] development is completely independent of VS.NET), browsing through the source code of the discarded applications clearly indicates that the lack of proper documentation is quite common.

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

Six Assemblies were chosen as the targets for this study. They span a range of different purposes and sizes. To help in the characterization of the Assemblies, a division into groups of similar purpose was created. This division is shown in Table 3.7.

Table 3.8 presents a summary of the eight assemblies chosen for this study, identifying their source application, and emphasizing their division into the groups in Table 3.7.

Table 3.7 – Group Characterization.

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

Table 3.8 – Assemblies used in the study.

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

NAnt [NAnt2008] is the .NET port of the Ant build tool; NDoc [NDoc2008] is an extensible code documentation generation tool for .NET. Due to the size of the applications, only the main assembly of each one was analyzed. Even so, exactly due to their size, they are representative of the rest of the code.

SharpZipLib [SharpZipL2008] is a .NET data archiving/compression library supporting all popular standards like Zip, Tar, GZip, BZip, etc; and CpSphere [CpSphere2008] is an implementation of the SMTP protocol which can be used to add mail sending capabilities to .NET applications. The first library is single-Assembly, and that Assembly is the target. For CpSphere, the main Assembly was selected as a target.

Finally, two of the .NET core platform Assemblies were chosen, mainly based on relevance (they are highly used) and documentation availability. Both Mono and SSCLI (codename Rotor) [SSCLI2008] were also considered as sources for the Infrastructure Assemblies. But, while in the first case the documentation style is different from that of VS.NET¹, in the second case, the examined source files (XML documentation is not included) contained no exception documentation.

These applications were chosen considering their popularity, number of users, complexity, bug reports, application support given by the developers and availability of source code. The application designers and programmers expertise for developing fault tolerant software could not be assessed. But, such attribute is not relevant in the spirit of this study. Otherwise, the study would be targeting the experiences of a small group of expert programmers and not the broader and more common programmer that has no special training in writing fault tolerant software but still uses the same tools (available in modern programming languages) as the experts.

3.3.3. Results

Table 3.9 summarizes the results obtained by running the Analyzer for the six targets, showing the percentage of documented and undocumented exceptions.

¹ Some of Mono's Assemblies include excellent exception documentation despite not using Microsoft style documentation tags. This just emphasizes the lack of agreement between different players and the fragility of this method.

Table 3.9 – Documented vs. Undocumented exceptions.

Group	Assembly	% Documented	% Not Documented
Applications	NAnt.Core.dll	3.4	96.6
	NDoc.Core.dll	0.5	99.5
Libraries	SharpZipLib.dll	21.2	78.9
	CpSphere.Mail.dll	8.4	91.6
Infrastructure	System.Runtime.Remoting.dll	16.2	83.8
	System.XML.dll	23.5	76.6
	Average	12.2	87.9
	Average for Applications	1.9	98.1
	Average for Libraries	14.8	85.2
	Average for Infrastructure	19.8	80.2

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

The results show that for the set of 6 different Assemblies over 87% of the relevant exceptions that the code can throw are not documented. For code directed mainly towards the end-user, this value goes up to 98%. For code aimed towards re-use by other programmers (libraries), it stands at about 85%. For infrastructure code, providing basic services for the .NET platform, this value is still as high as 80%.

Table 3.10 discriminates the types of exceptions that in the previous table are mentioned as documented. Thus, it offers insight into the types of exceptions that programmers are most likely to document.

Table 3.10 – Types of exceptions most likely to be documented.

Group	Assembly	% ILLs	% MCs	% Ts	% RTs
Applications	NAnt.Core.dll	0.0	24.5	73.5	2.0
	NDoc.Core.dll	0.0	0.0	100.0	0.0
Libraries	SharpZipLib.dll	4.6	11.4	84.1	0.0
	CpSphere.Mail.dll	0.0	17.2	82.8	0.0
Infrastructure	System.Runtime.Remoting.dll	0.0	100.0	0.0	0.0
	System.XML.dll	24.4	41.9	33.7	0.0
	Average	4.8	32.5	62.3	0.3
	Average for Group A	0.0	12.2	86.7	1.0
	Average for Group L	2.3	14.3	83.4	0.0
	Average for Group IS	12.2	71.0	16.9	0.0

The four most interesting graphs are also shown in the next figure (Figure 3.17).

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

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

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

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

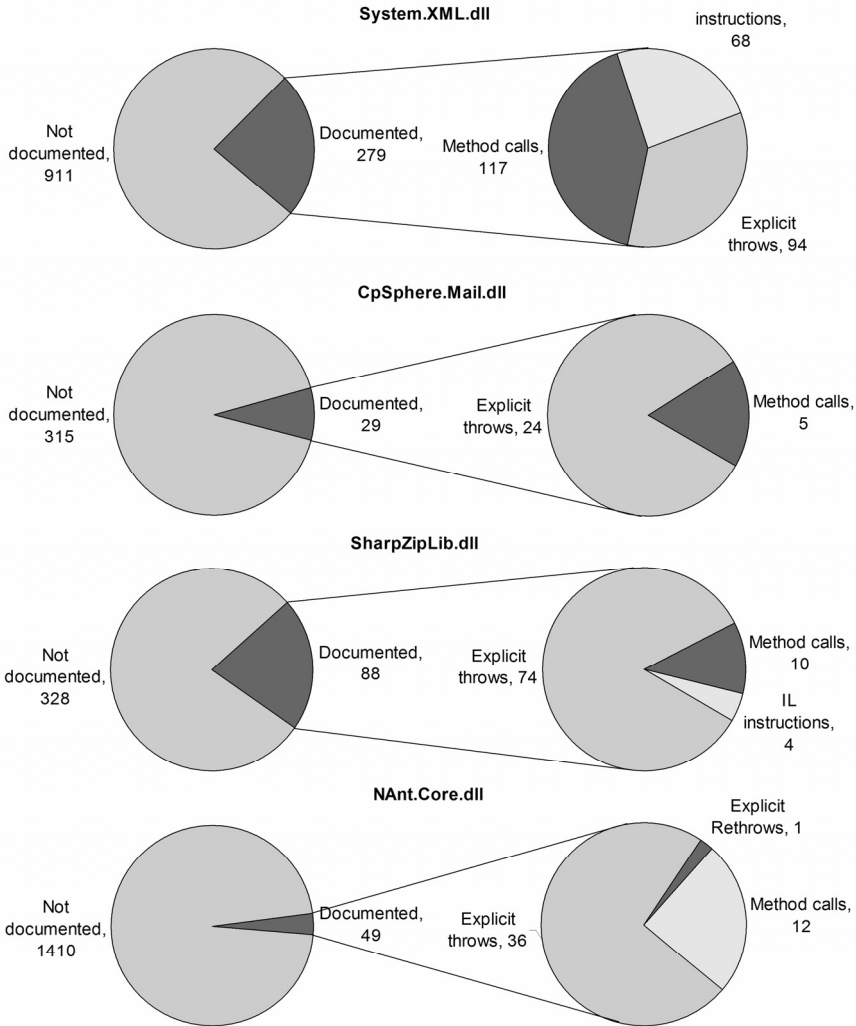


Figure 3.17 - Documentation of exceptions in four different assemblies.

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

possibly misleading anyone using his code as to the circumstances in which an exception occurred).

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

To solve this problem, the concept of *suspects*, mentioned in the previous section, was created. Suspects represent cases where we can state for sure that programmers could have done a better job on documenting their code. There are two types of suspects: code suspects and documentation suspects. Code suspects represent uncaught exceptions, detected in the code analysis and not in the documentation analysis, not originating from ILIs and that were found in methods for which there is documentation. Documentation suspects were a completely unexpected finding. They represent exceptions that the programmer marked as possible to occur in his code but were not detected in the code analysis and are, therefore, impossible to occur. Prior to running the Analyzer, we did not expect to find any documentation suspects, but Table 3.11, summarizing the results, shows that we still did find some of these cases.

Table 3.11 - Suspects for all eight Assemblies.

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

The documentation suspects we found are all, without exception, due to a specific feature of .NET – *properties*. Properties are internally implemented in .NET as one or two methods (depending on whether the property is read-only or not), a `get_<Property>` method and, possibly, a `set_<Property>` method. But the documentation tags only allow documenting a property as a whole (the internal methods are completely transparent to the

programmer). This carries more than one consequence. First, it means that if the documentation is not specific enough (it can explicitly say that the exception occurs only in setting the property), the programmer cannot know if the exception occurs in the getting or the setting of the property. Second, it renders attempts to do automatic exception handling or exception analysis like ours even more difficult, because it is almost impossible to have the computer read and interpret what someone wrote. We chose to have the Analyzer signal all these cases as suspects.

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

Finally, it is possible to compare the numbers presented in Figure 3.17 and Table 3.12 to get an estimate of the proportion of undocumented cases that are due to the plain absence of documentation. For this, we can take the number of undocumented detections from Figure 3.17 (joining in the values for the other 4 assemblies) and the number of code suspects from Table 3.12. The results of this comparison are shown in Table 3.13.

Table 3.12 – Type of detections responsible for code suspects.

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

Table 3.13 – Proportion of detections due to lack of documentation.

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

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

3.4. Summary

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

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

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

Exception handlers are not specific enough to deal with the detail of the occurring errors. The most preferable behavior is logging the problem or alerting the user about the error occurrence and abort the on-going action. Empty handlers, used to “silence” exceptions, will frequently hide serious problems or encourage bad utilization of programming language error handling constructs. Other detected problems, like the duplication of code between handlers, and the mingling of business code with exceptions handling code, among other problems are still to be tackled and represent an important research target.

This chapter also shows the magnitude of the problems of documentation absence and documentation quality in several .NET applications. More emphasis was put on the problem of documentation quality but Table 3.13, together with the great difficulties found when collecting Assemblies for this work, are a testimony of the problem of documentation absence.

Regarding documentation quality we found out that, on average, 87% of relevant exceptions thrown are not documented. These values range from around 80% to almost 98% growing as the amount of expected re-use declines. For the cases where it is possible to state for sure that the existing documentation is lacking in quality, most of the missing data is related to insufficient accounting of the exceptions that can occur by calling other methods, the rest being explicit throws, which are fairly well documented. This fact indicates that there may be benefits in developing ways of somehow automatically documenting methods by following call chains looking for the exceptions that may be propagated.

Ultimately, this study brings us to the checked vs. unchecked exceptions discussion. Why? Because checked exceptions are a means of getting exception information directly from a method, not having to manually go through all the chain of calls looking for exceptions, which is one of the downsides of unchecked exceptions. If the exception information associated to the method is accurate (which is very likely, because it is a compile-time check), programmers have one less excuse for not documenting their code. Furthermore, checked exceptions give us the possibility of providing techniques like automatic exception handling and even automatic code documentation.

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

The use of the unchecked model in .NET, and the lack of proper documentation about exceptions in these applications, can be seen as two of the causes for the fact that most of the exception classes used as catch arguments are quite generic and do not represent

specific treatment of errors, as one would expect. The lack of information about the exceptions that a method may throw can lead the programmer to perform a “catch-everything and do nothing” approach when facing problematic method calls.

The results discussed in this chapter lead us to the conclusion that, in general, exceptions are not being correctly used as an error handling tool. This also means that if the programming community at large does not use them correctly, probably this is a symptom of a serious design flaw in the mechanism: exception constructs, as they are, are not fully appropriate for handling application errors.

One may argue that the results would have been different if the programmers had been educated in the development of reliable software. But, this would not represent the broader community of developers. It is not even viable to assume that the large majority of developers worldwide will ever be educated in such way.

We believe that more work is needed on error handling mechanisms for programming languages. Modern exception handling constructs are the result of more than 30 years of research in the area. Also, any programmer that tries to develop code in a programming language, such as C# or Java, is forced to use exceptions. Programmers do know how exceptions are supposed to work and should be used. It is not the lack of this knowledge that leads them to write catch blocks that simply silence exceptions or log the problems. There are other reasons for justifying this practice and some of them can be related with the EH mechanism itself. The current exception handling mechanisms may indeed have the necessary semantics for being able to deal with problems, but if they are too cumbersome for the huge majority of developers to use them correctly, then these mechanisms must be revised. Future exception handling mechanisms should encourage the programmers to adopt best practices and use sound exception handling patterns.

In our work we approach the problem by trying to create automatic exception handling for the cases where “benign exception handling actions” can be defined (e.g., compressing a file on a disk full exception). In general, we try to free the programmer from the task of writing all the exception handling code by hand, forcing the runtime itself to automatically deal with the problems whenever possible. A complete description of the technique is discussed in the next chapter.

Automatic Exception Handling: A Proposal

In this chapter we present our automatic exception handling model. We will briefly introduce the motivations for our work, present the model's architecture, features, and programming model.

We will discuss how this model makes object-oriented software development simpler, quicker, cheaper, and, at the same time, how it is able to elevate the overall reliability of programs.

We conclude by showing in what aspects our model relates with its predecessors and in which it represents an innovation.

4.1. Introduction

In Chapter 2 we discussed the current state of the art in exception handling. We have assessed existing exception handling mechanisms strengths and weaknesses, discussed what could be done to improve current approaches, and set the guidelines for the development of an improved model. In Chapter 3 we discussed how the limitations inherent of existing exception handling models are affecting the way programmers use them when writing programs. Our conclusions on are quite alarming – *programmers are not correctly using exception handling mechanism for performing error recovery.*

Other authors have also questioned the programmers willeness and qualifications to write good exception handling code (e.g., [Shah2008a]). Based on our experience and the referenced work, it is safe to assume that many software developers consider writing code to deal with abnormal situations a dispensable task that only diverts them from their main objective: writing the application’s business logic code. Furthermore, we have witnessed a shift in how exception handling is perceived – “developers have shifted their perspective on exception handling from the intended proactive approach (i.e., how to handle possible exceptions) to a reactive approach (i.e., using exception handling as debugging aids).” [Shah2008a]

In some sense, we agree that exception handling in today’s applications has become a cumbersome task. There are thousands of possible exceptions types for developers to deal with when writing software. Each application (or software library) introduces its own types of exceptions and, many times, it is not clear what the problem behind an exception occurrence is. Also, error handling code is scattered along the entire program’s code and mingled with business logic, making control flow difficult to track in the presence of errors. Another problem is that the rules enforced by compilers and run-time platforms for checking the code safety are (sometimes) considered intrusive by developers because they force them to alter their code writing style. Last but not least, testing of exception handling code is not a simple task.

The unwillingness of software designers to deal with exceptions correctly and follow some well known best-practices for exception handling [Wirfs-Brock2006], undoubtedly, contributes to the lowering of the quality of programs and their resilience to errors. It is obvious that something is not right with current exception handling models: they are not adequate enough for developers.

We claim that, in many situations, a platform level automatic system capable of providing benign recovery actions for exceptions would achieve better results in terms of reliability than the programmers' exception handling code. An exception handling mechanism should provide effective exception handling and not lower the productivity of programmers. This may seem a tall claim but, considering that for a large number of exception types it is possible to have the runtime providing a set of benign recovery actions that automatically recover the system when an exception is raised, the problem becomes treatable. The case for Automatic Exception Handling (AEH) [Cabral2006,Cabral2008] is that, for the majority of cases, the programmer should not have to write exception handling code. Benign recovery actions should be part of the runtime platform and should be automatically executed when an exception is raised. By doing so, the programmer is freed from the "burden" of writing exception handling code for a large number of situations. In a sense, automatic exception handling should work as a Garbage Collector for exceptions. Without, or with minimal, programmer intervention, the mechanism should automatically execute benign recovery actions for the exceptions being raised in the running code.

We propose an exception handling model where the most common exceptions are dealt automatically by the runtime environment without forcing the programmer to write any code. To be viable, our model must effectively lead to less code being written by the programmer, while at the same time allowing the development of more robust applications.

The proposed system is based on a *Software Transactional Memory* (STM) [Shavit1995] approach for maintaining state consistency while benign *recovery blocks* [Horning1974] are tried for recovering the application (*backward error recovery*).

4.2. The Model

In this section we discuss the core model of an automatic exception handling system using a transactional approach. For a question of easiness and readability, the discussion uses the Java exception model and language notation. Nevertheless, automatic exception handling is applicable and readily adaptable to any modern managed programming language environment (e.g., C#/.NET, Python, etc.).

```
Filewriter file = null;
try
{
    // Open file
    file = new Filewriter("data.txt");

    // write some data into it
    for (int i=0; i<1024; i++)
        file.write("Here's the data: " + i);
    // Close file
    file.close();
}
```

Listing 4.1 – Writing to a file in a transactional try block

To understand the programming model for automatic exception handling, consider the following grammar fragment, adapted from the Java Language Specification [Gosling2005]:

```
Statement ::=
    try Block ( Catches | [Catches] finally Block )
    try Block
    ...
```

We introduce a simple modification, shown in bold: try blocks do not need to have a catch handler, which becomes optional.

When the application is deployed it contains a number of benign recovery blocks, configured by exception type, which may be executed when an exception occurs. These recovery blocks can be shipped directly with the virtual machine, or correspond to custom code shipped with the application. The system can execute multiple recovery blocks for each exception occurrence inside a try block. After the execution of each recovery block, the code inside the try block is re-attempted. A transactional system ensures that the effects of a failed try block are discarded prior to the execution of the recovery blocks and the try block own re-execution.

Going back to the example of Listing 1.2 (page 12), the programmer would only have to write the application logic code, that is, saving the data into the file, enclosing it in a try block. This is shown in Listing 4.1. If an exception is raised, the runtime system provides the necessary benign recovery actions for trying to solve the problem. Each recovery action is executed once, after which a new re-execution of the faulty try block is attempted.

4.2.1. Benign Recovery Actions

The central argument of this model is that for a large number of abnormal situations, the runtime system should be able to deal with the problem without having to force the programmer to write code. In fact, in many situations the runtime should be able to provide better solutions than what the programmer would, since most programmers do not focus on error handling but on writing application logic.

Consider an example like writing a file to disk. During that operation, a `DiskFullException` can occur. Instead of directly throwing an exception, for which the programmer has to explicitly provide an exception handling block, the runtime system could benignly try different recovery operations. For instance:

1. Remove temporary files on disk;
2. Compress not frequently used folders;
3. Reduce the size of the swap file;
4. Move selected files to a remote server;
5. Mount an extra disk;
6. Notify the user asking for help on the problem.

A key point of these actions is that they are benign. They do not affect the environment on which the program is executing in a harmful or potentially destructive way. A counterexample of a non-benign action would be to automatically erase some user files to make space available.

Similarly to the six recovery actions for `DiskFullException` defined above, it is also possible to define them for a large number of platform-level exceptions. For instance, if a network connection is broken: the system can try to reconnect to the server automatically; try to connect to a different server (or servers); try to use a different network interface; and so on. If an update on a transactional database fails, it is possible to try to re-execute the transaction. If an authentication process fails, a different authentication module can be tried, etc.

For sure, application-specific exceptions can only be dealt explicitly by the programmer. But for a number of common application exceptions, benign recovery actions can be

provided¹. These actions can be made available *off-the-shelf* with the runtime environment and configured when the application is either developed or deployed. Notice that the objective is not to provide automatic exception for all types of exceptions. Instead, the goal is to provide automatic handling for the most common types of exceptions associated with a particular development platform, substantially easing the life of the programmer.

In our model, recovery actions are defined at platform level and shipped with the platform itself. These default recovery actions will be associated to the exception types defined on the platform and on the system's libraries. As an example, let us consider which recovery actions could be provided by default by the Java virtual machine for the `NoRouteToHostException` or the `PortUnreachableException`:

1. Rollback the failing `try` block and re-execute the instructions inside the protected block;
2. Rollback the failing `try` block and re-execute after pausing for predefined time period;
3. Rollback the failing `try` block and connect to a different host and re-execute. A location for the alternative connections can be provided by the programmer once or every time;
4. Notify the user of the problem and provide details on the exception cause, allowing him to manually correct the problem. Afterwards, rollback and re-execute the `try` block;
5. Re-throw the exception and propagate upper the call stack.

But, independently of the form how recovery actions are built into the platform, the fundamental rules that a system has to follow to implement the automatic exception model are:

1. The code of recovery actions must be available for execution from any location on the run-time environment;

¹ In the next chapter we will assess to which degree benign exception handling actions can be defined for a large set of platform specific exception types.

2. Recovery code must be able to control the rollback of the code raising the exception. In some situations, it may be desirable not to rollback the failing code. Thus, the transactional mechanism controls must be accessible from inside the recovery blocks;
3. It must be possible to pass values and object references (memory references) from the location where an exception is raised to the recovery action code. By doing so, system designers will be able to provide more powerful benign recovery actions;
4. Recovery functions do not need to change the code inside the protected region where their activation took place but they must be able to change the execution environment state where that code will re-execute;
5. Exceptions occurring inside a recovery action must be handled locally or propagated to the upper transaction or element in the call stack;
6. The transaction where the exception that activated the recovery action was raised is aborted before the execution of the recovery actions and the new transaction begin at the start of the recovery code;
7. The runtime environment must be able to control the serial execution of the different recovery actions for an exception occurrence. It must ensure that an action is not repeated (executed more than once) for the same occurrence, that all actions predefined for that abnormal event are executed until the exception manifestation is eliminated, and that the adequate actions are bound to the correct exceptional events.

Furthermore, to improve performance and avoid the execution of inapplicable recovery actions, recovery actions can be bound to exception occurrences in several forms:

1. Globally, by exception type (default);
2. By exception type and application (program, software component, package, assembly, etc);
3. By exception type and class (i.e., the class where a certain exception type can be raised);

4. By exception type and method (i.e., the method and class where a certain exception type can be raised);
5. Specifically to a try block.

Customizing the binding of recovery blocks to particular exceptional occurrences is achieved through the use of configuration files.

4.2.2. Programming Model

In terms of the programming model, the system works as follows:

1. While writing code where exceptions can be raised, the programmer demarks them with a normal try statement (or block). The catch part is optional;
2. If a catch block is present, this means that the programmer wants to explicitly handle the exception. The provided code is executed when a corresponding exception is raised. This corresponds to the *normal* exception handling model;
3. If a catch block is not present, then the corresponding try block is a candidate for being handled automatically by the runtime system;

At run-time:

1. Every time a try block is reached, a new transactional context is created. All object and variable accesses are recorded inside the block and, later on, can either be committed or aborted. Nested try blocks correspond to nested transactions.
2. On exiting a try block normally (i.e., no exception occurred), the transaction commits.
3. If an exception occurs during the try block and if a suitable catch block exists, either in the current method or above, the flow of execution passes to that block. This corresponds to the *normal* exception handling model. Note that before passing the flow of execution to the catch handler, the execution commits making visible the modifications that occurred on the try visible. It may also be necessary to perform stack unwind if the exception is being propagated.

4. If an exception occurs within a `try` block with no `catch` statement or without a suitable `catch` statement¹, that block is a candidate for automatic exception handling. If a suitable benign recover block exists (i.e., of the correct type), automatic recovery is tried. If not, the transaction commits and the exception is propagated as in the *normal* exception handling mechanism.
5. For performing automatic exception recovery, the system rolls-back the current transaction. That ensures that the application is in a clean state. The runtime then tries to execute each of the configured recovery blocks, one-at-a-time. After each recovery block is executed, the `try` block is re-executed from the beginning. This happens until either the execution succeeds or all options are exhausted. At that point, according to a deployment configuration file, either a “Log&Abort” operation is executed, which terminates the program, or the original exception is re-thrown at the offending statement.

Since a `try` block can be executed multiple times, a critical aspect of this framework is that they (`try` blocks) must be transactional. This means that after a recovery block is executed, the application state must be automatically restored to its condition as of when the block was first entered. This aspect will be discussed in the next section in greater detail.

Going back to the example on Listing 4.1, if a `DiskFullException` is thrown on the `file.write()` operation, the first recovery block is executed and the `try` block re-executed from the beginning. If on the second execution an exception is still thrown, the second recovery block is tried. This happens until either the execution succeeds or all options are exhausted leading the program to be terminated or the original `DiskFullException` being re-thrown. Figure 4.1 illustrates this mechanism.

The information that associates exception occurrences with concrete handling actions is kept on a runtime system configuration file which is shipped with the runtime platform. Nevertheless, the system leaves room for customizing its behavior. Other configuration files can be released with the application itself, specifying what happens for each exception type across the program. It is also possible to specify what happens for each class, method, or specific `try` block. Possible actions include: i) execute a number of recovery actions before throwing an exception; ii) directly throwing the exception assuming that it will be caught at a higher level in the stack, or eventually abort the program.

¹ This happens if the existing `catch` statements do not correspond to the exception type being thrown.

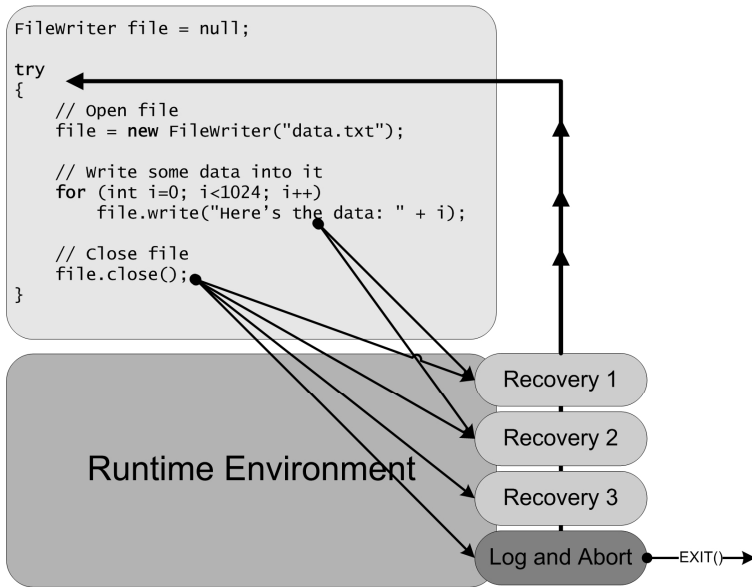


Figure 4.1 - The runtime system provides recovery.

4.2.3. Transactional System

As it was previously mentioned, a key point of the whole approach is that try blocks are executed transactionally. When repeating the execution of a try block it is essential to guarantee:

1. The application state (variables, data structures, etc.) is as it was on the first execution of the block;
2. The execution of the try block is isolated. This means that if external I/O operations are executed, it must be possible to undo them, or they must be idempotent (i.e., can be repeated).

These important aspects are discussed next.

Restoring Application State

Restoring application state is vital in order to guarantee that the intent of the programmer is preserved. For instance, if a programmer is incrementing a variable `total` inside a try block, if the block is re-executed due to an exception, after exiting the block the variable must only have been incremented once. That is to say that the program semantic should remain the same independently of the recovery system in place.

For restoring the application state several approaches are possible. Typically methods include:

1. When entering a protected block, create a copy of all touched objects. These copies are used whenever updates are made. If the block exits normally, the original objects are replaced by the updated ones. If a re-execution takes place, the copies are discarded, being the original objects preserved.
2. No copies are made. Instead updates are done in place. All changes to objects are logged. If a block exits normally, the logs can be discarded. If a re-execution takes place the logs are used to restore the objects to their initial state.

The details regarding each method are actually somewhat tricky to implement and incur in different overheads. For instance, in object-oriented platforms, the first approach feels more natural, though it may require more memory than the second one and rely heavily on the execution of the garbage collector.

Suffice to say that this transactional type of system actually corresponds to having a Software Transactional Memory (STM) framework [Shavit1995] in place where the `atomic` keyword corresponds to a `try` block. Though the STM model is essential to our system, its definition is not the core of this proposal. There is currently very active research on how to optimize such systems and make them available on the mainstream. In our case we not only benefit from those systems becoming available (e.g., *AtomJava* [Hindman2006], *Atomos* [Carlstrom2006]), but also the associated overheads for exception handling can be much lower. In general, we expect `try` blocks to succeed, thus the system can be heavily optimized in that direction.

As a general guideline, an STM system for usage with our model must support, at least, *closed nested transactions* [Moss2006].

Isolation in `try` blocks

As it was mentioned, preserving only application state is not enough. For instance, on the example of Listing 4.1, if the `FileWriter` class (or the runtime system) is not aware that a transaction is taking place, data can end up being written twice into the file. What this means is that the transaction is not isolated from the exterior. It leaks information.

For dealing with this problem, as a general rule, if an unprepared class (i.e., non-transactional aware) is detected inside of a try block, the compiler forces the programmer to write a catch handler. This means that for legacy code, the programmer is in the same situation as today – the runtime system cannot help him. Also, some I/O actions cannot ever be undone (e.g., consider the case of “firing a missile”). Again, classes that encapsulate that kind of operations are not candidates for automatic exception handling recovery. If exception handling is involved, the programmer must explicitly deal with it. Nevertheless, transactional and non-transactional code can coexist in the same program but, the transactional code cannot reference non-transactional classes and methods.

Classes that can be made *transaction-aware* must somehow be recognized by the runtime system. A simple approach for this problem is making them implement a `Transactional` interface. Transactional-aware classes must explicitly provide at least three callback methods on this interface: one for the runtime system to signal that a transactional context is being entered; one for the runtime system to signal that the changes are to be committed; and one for the runtime system to signal that the operations have to be undone. Since try blocks can be nested, an identifier for each transactional context must also be passed.

Although using this approach may apparently seem complicated or cumbersome it is not so in practice. For once, only classes that perform external I/O operations must be marked in this way. Ordinary objects are already automatically made transactional by the normal STM system. The complexity of implementing the callback methods varies, but their semantic is clear.

In terms of the classes that can be made transactional, several alternatives exist:

1. If I/O is involved, it may be possible to temporarily buffer it, until a commit is possible;
2. In some cases, it is possible to create idempotent operations for the class allowing operations to be repeatable.

Considering once again the `FileWriter` class, an example of the first approach would be to write the data into a temporary file. If the try block commits, the temporary file could be renamed to the correct name. An example of the second is actually the `FileWriter` as implemented on Sun’s JDK, and as it is used on this example. When the `FileWriter` constructor is executed it creates a new file on disk, truncating any existing one, the try block can be executed without any side-effects (i.e., in an idempotent way). However, this

would not be case if the `FileWriter` constructor, which supports appending to the file, would be used.

On a final note, it should be mentioned that the same problems that occur in database systems and STM systems can also take place. For instance, in the context of a transaction, reading data previously written can lead to problems. Consider the case where inside a `try` block a `FileReader` tries to read data that has just been written. Since the writer has not yet committed, the reader may not be able to read it on a naïve implementation. There is currently no general solution for those problems. The developer of these classes must take care in preventing them. In most cases it is simple to provide hooks for undoing the operations or detect conflicts with other operations. Also, in a general development platform, the number of classes that are directly involved with external I/O is limited. In any case, if supporting a certain class proves to be too hard, the platform provider can always opt for making it non-transactional. In that case, the application developer has to explicitly write exception handlers for `try` blocks that use it.

Nested `try` blocks

Our model allows the nesting of `try` blocks. This is the same as saying that the model implements *closed nested transactions*. Changes made by the nested transaction are not visible to the parent transaction until the nested transaction is committed. This follows from the isolation property of transactions. But, the most important aspect related with nested transactions in our model is the definition of what happens when an exception is raised inside one.

Exceptions raised inside a protected block will be handled by the system preset recovery actions or by an explicitly designed catch handler. If the handling code is not successful on eradicating the exceptional occurrence, the exception is re-thrown and passed to the upper transaction (upper `try` block on the call stack). At this point the process repeats itself: the exception will be handled by the system preset recovery actions or by an explicitly designed catch handler, the current `try` block can be re-executed (an undefined number of times) and as a consequence the inner transaction is also re-executed. If the exception does not manifest itself again the inner transaction can attempt to `commit` and its results become visible for the parent transaction. If the exception does not disappear, the parent transaction will ultimately fail and the exception re-thrown to any upper transaction or to the runtime system causing the program execution to terminate.

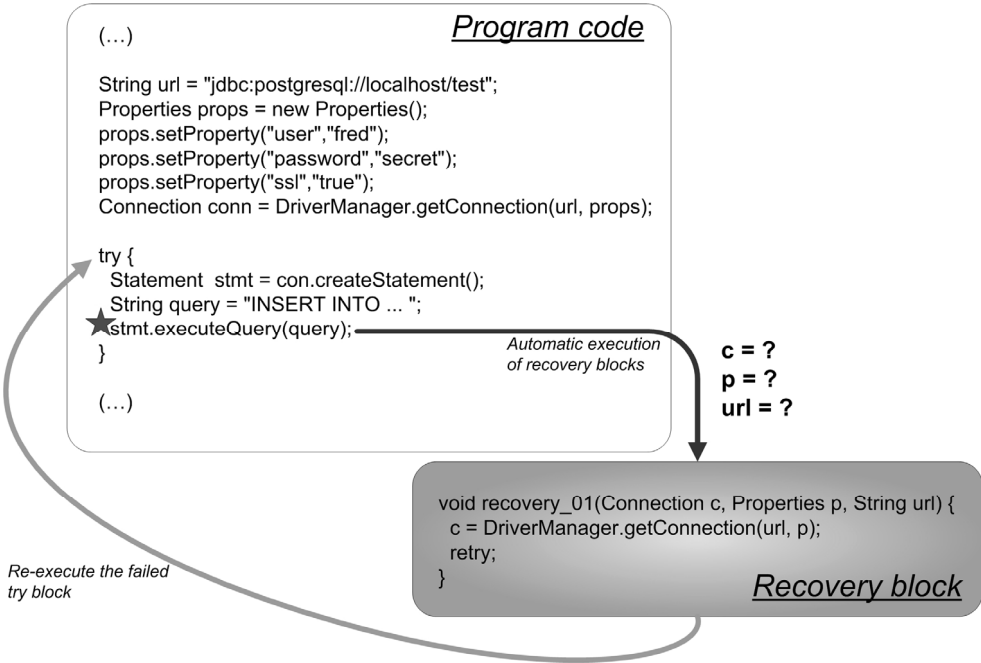


Figure 4.2 - Passing parameters to recovery blocks.

4.2.4. Exception Parameters

To provide automatic recovery actions that can, in fact, help recover systems in the presence of a large number of distinct errors, it is necessary to implement a system capable of efficiently communicating values and object references from the location where exceptions are raised to recovery blocks. In the example of Figure 4.2, in the event of an exception being raised inside the try block, one or more recovery blocks might attempt to recover and request the re-execution of the failed try block. In the example, the recovery block needs three input parameters. But, since the recovery code is shipped with the platform it is difficult for the programmer to define which references should be passed.

In simpler cases, the selection of references and values to use as parameters for the recovery blocks can be made automatically, and the necessary references held by the exception object itself. In more complex computations, a more demanding list of parameters will be necessary. In these cases, it would be very difficult for the system to automatically define which references to use. Thus, in such cases, the system will require human intervention.

For selecting the variables in the code which will be used as parameters for the recovery actions, and considering that one of our main objectives is to minimize the work of programmers in the development of exception handling code, we opted on using configuration files instead of creating new language constructs. Using language support for defining the parameters for recovery actions would only be viable if the code using such constructs is located near to the exception raising point. With our approach we avoid scattering the exception handling code through out the entire program code.

On the other hand, defining configuration files (e.g., such as the XML formatted files we will present and discuss in the next chapter) can be a hard and cumbersome task. It can even raise code visibility and readability concerns when a developer needs to know the control flow of its program in the presence of an exception. We expected such problems to occur and we believe that the solution is in the automatic or semi-automatic generation of the configuration files and in resorting to visual programming aid tools.

As an example, we propose the incorporation of an exception alert and configuration system into source code editing program in such a way that:

1. When a programmer is writing the application logic code, whenever he or she writes the code to invoke a method, the development environment automatically verifies the exception list for the called method and alerts the programmer (using a visual callout for instance) if that method can raise an exception. The system emits the alert when the problematic method call is not inside a try block and the exception type is not on the list of exceptions being raised on the calling method. The alert will also be issued if the call is inside a protected block, the exception is not on the exception list, the system does not possesses benign recovery actions for that particular exception type and the programmer does not defines a catch block for the exception.
2. If the called method is inside a try block and the system has automatic recovery code for the exceptions being raised inside, the system verifies the need for selecting parameters for the recovery methods. If the parameters cannot be automatically set by the system, the programmer is informed by a visual callout that he or she must select the adequate list of variables to be used as parameters for the exception handing process. If the programmer opts for defining those parameters immediately, the system can provide him with a window interface for doing so. In this visual interface, he or she is able to know which exception

type is to be automatically handled by the system and which class implements the recovery actions and which actions are to be tried. At this point, the developer can select the variables in the code that will be passed to each recovery method. The configuration files for the application are based on the introduced data and automatically generated.

The way systems actually make the selected variables and references available to the handling code can vary for each implementation. In our test platform, we used a hybrid approach: in some cases the code for the handling actions was *inlined* with the code causing the exception; and, in other situations, the variables (their values or references) were made available on the stack of the recovery method. Short recovery procedures were *inlined* to avoid heavier performance penalties by cutting the extra-costs involved in method invocations.

4.2.5. Exception Handling Model Features

Garcia et al. [Garcia2001] defined ten different aspects through which an exception handling model can be characterized. We will use the same notation to describe the automatic exception handling model:

Exception representation

As we have already discussed in Chapter 2, exception handling mechanisms can have different structures for representing exceptions: *symbols*, *data objects*, and *full objects*. Objects guarantee the uniformity of the programming paradigm; objects do not require the usage of extra global variables for passing information to the exception handler, thus benefiting modularity.

In our model, in resemblance with what happens in languages such as C++, Java, C# and Delphi, we use data objects to represent exceptions. Being data objects, exceptions do not implement their own functionality and thus require special support from the runtime. We provide a special keyword (`throw`) to allow the programmer to raise exceptions.

External exceptions in signatures

Exception lists are mechanisms that allow a method to declare to the outside world which exceptions its actions can produce that are not being handled inside, thus being propagated to the outside. Systems implementing checked exceptions make exception lists

mandatory while unchecked systems make its usage optional. Exception lists are, in our perspective, an essential feature for developing highly dependable software since, as we have observed in prior work [Cabral2007b,Sacramento2006], programmers cannot be trusted to document their exceptions.

In our model exception lists are mandatory. Nevertheless, we expect that in most cases the handling of the potential exceptions will be done by the platform and not by the programmer's code, thus, making our checked exceptions model much less demanding for the programmer than its predecessors. Most times the declaration of a `try` block will suffice.

Our model guarantees that the programmer is aware of the potential problems that his code will face, but, at the same time, tries to prevent the programmer from being tempted to fall into some less recommended programming practices such as silencing exception, or log and terminate on non-fatal exceptional occurrences, because he or she is not forced to provide handling code for all exceptions.

Separation between internal and external exception

In some situations, it can be useful to differentiate which exceptions are to be handled inside a method and which are to be dealt with outside the method. Such feature requires a special support by the programming language that must provide different ways to raise both kinds of exceptions.

In our model, we do not provide any special means to implement such feature. We believe that object-oriented exceptions can mimic this functionality. It is possible to use object-oriented inheritance to create new types of exceptions so that these new types can be handled differently: the descendents of one type can be treated as internal exceptions, while the descendents of the other type can be treated as external exceptions.

Attachment of handlers

We have also mentioned in Chapter 2 that the definition of the protected region to which an exception handler is associated can differ in many aspects. For example, a handler can be associated with (i) a statement, (ii) a block of statements, (iii) a method, (iv) an object, (v) a class, or (vi) an exception class.

Garcia et al. argued against the usage of block handlers. From all association kinds the authors consider this the weakest type. The authors defend that “the use of block handlers violates explicit separation of concerns, since exceptional code is intermingled with normal code albeit moved to the end of the block”. Most times, blocks of statements are defined with the sole purpose of attaching a handler. This practice can lead to the development of software which is difficult to read, maintain and test.

Our model supports all kinds of attachments, including block handlers. But, we take things a step further, in the sense that we allow the attachment of multiple recovery blocks to the same entity (or even code block). In the case of an exception occurrence, one or all the recovery blocks can be executed.

Handler binding

There are three approaches¹ for discovering (identifying) the handler that should be executed when an abnormal situation is detected: the *static approach*; the *dynamic approach*; and the *semi-dynamic approach (hybrid)*. The hybrid approach mingles the two previous techniques: a handler can be statically associated with an exception occurrence but, in the event that no suitable handler is found on the immediate lexical context, the runtime system is responsible of dynamically selecting the appropriate handler.

Our approach is hybrid. In our model it is possible to provide lexical association of handlers, perform dynamic search of handlers up the call stack at run-time, and associate recovery actions with exception types (at program-, class-, method-, or block-level) based on configuration data available (and changeable) at run-time.

Propagation of exceptions

Garcia et al. identified two design solutions for exception propagation: *explicit propagation*; and *automatic (implicit) propagation*. The first kind only allows the propagation of the exception to the immediate caller of the failing operation. The second kind of propagation, allows exceptions to be transmitted through multiple levels on the call stack until a suitable handler is found or the program is terminated.

We opted for the automatic approach since it requires less intervention (and less code) from the programmer. Nonetheless, this approach has its shortcomings. Automatic

¹ Please consult Chapter 2 for more detailed information.

propagation allows exceptions to be transmitted through multiple levels on the call stack until a suitable handler is found or the program is terminated. The problem is that there are no guarantees that an exception occurrence will be bound to the most appropriate handler and, at the same time, the propagation of an exception through different levels of abstraction can cause the unexpected exposition of implementation details and the corresponding degradation of encapsulation and modularity.

On the other hand, using checked exceptions helps minimizing some negative aspects: all methods in the call stack are aware of the exceptions being propagated; implementation details can remain hidden by transforming internal exception type into different types for the communicating to the exterior; and different exception types can be associated with different components thus maintaining some degree of modularity and re-usability. In our case, exceptions are not only propagated in the call stack vertically but also in a transversal way to the recovery blocks code.

Continuation of the control flow

There are two different propagation models that delineate where the normal flow of execution is resumed after the execution of an exception handler: the *termination* (simple and *retry*) and *resumption* models. The termination model has simpler linguistics and does not require multiple kinds of signals for raising different types of exceptions. The resumption model introduces a new level of indirection, not only the *caller* of a function is dependent of the invoked function, but also the *callee* becomes dependent of the *caller* when an exception is raised.

The termination is the best option for our model, both in terms of software reliability and simplicity. Besides the traditional control flow, where execution is transferred to a handler and continues in the normal path after the handler completion, the principal kind of control flow in our model involves transferring execution to a set of recovery blocks made available by the platform itself, restoring the application state using transactions, and the re-execution of the protected region of code until the exception stops or there are no more recovery blocks to try.

Clean-up actions

An operation will either terminate correctly or with errors. In both cases, it is important that the program state remains coherent. Clean-up actions allow the program to recover to a valid state, or undo the effects of some actions.

Our model allows the usage of specific construct (similar to Java's `finally`) to perform clean-up actions and, at the same time, provides an automatic mechanism (software transactional memory) that allows the effects of the code inside a protected region to be rolled back in the occurrence of an exception.

Reliability checks

Our model uses both *static* and *dynamic* checks. The first kind of reliability checks (the larger set) are performed by the compiler while the second kind is performed by the runtime system. These checks must verify the correct utilization of the checked exception model. At run-time, checks are reduced to the operation of finding the most adequate handling code for the raised exceptions and raising an error if none is found.

Concurrent exception handling

Concurrent systems offer a completely new set of challenges in terms of exception handling. It is difficult to design, analyze, modify and, sometimes, understand concurrent object-oriented systems. Thus, in many situations it is not possible to guarantee that erroneous information is always contained inside an object. In such systems, and in the presence of an abnormal situation, we will most probably have to deal with several interconnected objects simultaneously.

At the present time we do not include any special way of automating the communication of exceptional information between concurrent handlers. It is extremely difficult to preview what the requirements, the architecture, the number of participants that a concurrent application will have at run-time. And, it is even harder to forecast what will happen in the presence of an abnormal situation. The complexity of the problem represents a real barrier for the design and implementation of a cooperating automatic exception handling mechanism. In terms of reliability it may be preferable to leave the coding of concurrent exception handling facilities on the programmers' hands. Nonetheless, the existence of a transactional mechanism in our model can help on cleaning-up the post-exception application state and in creating "virtual" checkpoints for recovery to commence. Furthermore, the simpler serial exception handling, that does not require communication with concurrent handlers, can still be dealt by the platform automatically.

```
void swap(Stack a, Stack b) {
    atomic {
        do {
            Object item1 = a.pop(); // Might throw exception
            Object item2 = b.pop(); // Might throw exception
            a.push(item2);          // Might throw exception
            b.push(item1);          // Might throw exception
        } or else {
            Object value = a.top().getValue();
            a.top().setValue(b.top().getValue());
            b.top().setValue(value);
        }
    } on failure {
        a.rejuvenate();
        b.rejuvenate();
        retry;
    }
}
```

Listing 4.2 – swap method using an `atomic` block and alternative execution paths.

4.3. Related Work

The initial proposal for our automatic exception handling model using transactions was first published in 2006 at the USENIX HotDep'06 workshop [Cabral2006]. Since then, our ideas evolved into a mature model. During this time, other authors, with similar motivation, have also advanced the state of the art in exception handling in directions analogous to ours. In this section we will discuss these new approaches and, whenever relevant, we will point out the similarities with our work or show how these proposals can be supportive of our model.

Recently, we encountered an article by Christof Fetzer and Pascal Felber [Fetzer2007] that explores the concept of using `atomic` blocks for improving program correctness. The authors suggest not using exception handling mechanisms to deal with errors and abnormal situations. Instead, they recommend using `atomic` blocks to delimit parts of the code where problems might arise. Inside these blocks, programmers can explicitly detect errors and perform the necessary corrective operations in order to abandon the failed `atomic` block or `rollback` and `retry` its execution.

Listing 4.2 illustrates the approach proposed by Fetzer and Felber. If the exchange operation fails in the `do` block, it is automatically retried by executing the `or else` clause that uses another strategy. If both fail, the stack objects are rejuvenated and the complete `atomic` block is retried (starting with the default execution path).

Though, this work presents perceptible similarities with ours, the bulk of the concepts on our model, such as automatic exception handling, the usage of recovery blocks, and the complete separation between exceptional and normal code, are not mentioned on the cited article. Furthermore, the authors do not provide any implementation or validation of their ideas. Their work is limited to the proposal.

Daniel Lanvin et al. [Lanvin2009] introduced the concept of *reconstructor*. A reconstructor is an extension of the semantics of common object-oriented languages to restore the previous consistent state of a system in the presence of error, avoiding some of the tasks that exception handling mechanisms delegate to developers. In the past, there have been several attempts to integrate some form of automatic object-state recovery into object-oriented systems [Cristian1979,Campbell1983,Oki1983, Plank1996,Tikhomirova1997, Silva1997,Garthwaite1998,Shinnar2004,Fetzer2004]. But, a solution is yet to be adopted by any mainstream programming language or platform. Lavin et al. explain that the costs involved on incorporating this kind of mechanism into a program, in a general way, are just too high. Memory consumption is usually the most serious problem because this kind of mechanism commonly relies on some sort of checkpointing. To lower memory consumption, Lanvin et al. proposed a way to restrict the number of objects that should be recovered and use specialized reconstructor methods to control the object's state recovery process.

There are two kinds of reconstructors, *implicit* and *explicit*. Implicit reconstructors are automatically generated for each attribute in an object. For each change made to the value of an attribute, and consequently to the state of the object owning that attribute, there will be a reconstructor method capable of undoing the modification. Lanvin et al. limited the creation of implicit reconstructors to accesses made through setter methods. Explicit reconstructors, are different from implicit reconstructors since they have a more ambitious goal than just recovering the state of an object. Explicit reconstructors are designed for recovering the consistency of a system. These reconstructors must be designed by the developer in the form of a *compensable* method. This method is responsible for performing the actions that will return consistency to the system. Compensable methods are invoked by a specialized object which is also responsible for managing the parameters that are passed to the method.

```

// Declaration of an implicit reconstructor
class ExampleClass {
    @Reconstructable private int counter;
    public void setCounter(int value) {
        counter = value;
    }
}
...
// Declaration of an explicit reconstructor for the
// prepareDelivery method
public void prepareDelivery(Type1 param1, ..., TypeN paramN) {
}
...
@Reconstructor public void __prepareDelivery(
    Type1 param1, ..., TypeN paramN) {
}
...
}

```

Listing 4.3 – Implicit and Explicit reconstructors declaration.

The code in Listing 4.3 is an adaptation of the examples provided by Landin et al. in [Lanvin2009] and illustrates the way a programmer declares implicit and explicit reconstructors. The authors use `@Reconstructable` to indicate a code pre-processor that an implicit reconstructor must be implemented. The pre-processor will introduce new code to implement the implicit reconstructor. There is also a `@Unreconstructable` tag for marking attributes that should not be reconstructed thus avoiding unnecessary overload. The `@Reconstructor` tag is used to define a method as being the compensable method for another method. The relation between a method and its reconstructor is defined by the application of a pre-defined naming convention. The reconstructor is able to receive the same arguments as the method that it is reconstructing, but it is possible to pass additional parameters by marking variable inside the first method as `@ToReconstructor`.

Lanvin et al. have to delimit the reconstruction scope. Each time something goes wrong, the reconstructors in the execution path are activated in the reverse order of their creation. If there was only one context, each time reconstruction was activated the program would return to its initial state. Thus, the authors associated the functionality of reconstructors with the scope of `try/catch` blocks as shown in Listing 4.4. As happens with transactions in our model, a context initiates at the start of a `try` block and is closed at the end. If something goes wrong, the state of the objects accessed inside the protected region is returned to what it was at the start of the context. Our model goes a step further: we not only allow the automatic rollback of the state of objects, without the need for specialized tags, but we also provide automatic recovery for whatever went wrong in the execution of

```
ContextHandler ctx = ContextFactory.createContext();
try {
    ...
}
catch (...) {
    ctx.reconstruct();
}
finally (...) {
    ctx.discard();
}
```

Listing 4.4 - Context management integration in try/catch blocks.

the try block. In terms of recovering the consistency of a system or undoing external nonidempotent operations our approaches are comparable. Both require the intervention of the developer to write the necessary code. Lavin et al. use compensable methods for this while we allow the developer to write handling for each particular abnormal occurrence.

In [Chang2009] the authors propose a methodology and self-healing technology that can reduce the occurrence of failures caused by common integration problems that are identified and documented by COTS developers. With this methodology, application developers inject *healing connectors* into their systems to automatically repair problems caused by misuses of COTS products. If something goes wrong in the execution of a method $m()$, when component A invokes $m()$ on component B, an exception is raised and passed from B to A. Healing connectors stand between the two components and intercept the exception. Next, the connectors try to remove the problem using any available healing strategies and re-invoke the failed method. If the method executes without throwing any exception, the result is passed to A, if not the exception is propagated to A. Healing strategies are defined by COTS developers.

The authors use Aspect Oriented Programming (AOP) to add healing connectors into programs. A healing connector is composed of three parts: the *connector* that intercepts exceptions propagated between components and represents the *aspect*; the *healing strategies* that are defined in classes and implement the healing actions or, in AOP terminology, the *advice*; and the *identification of the locations where to inject the code*, also known as *pointcuts* and *joinpoints*. Much of the information used for defining code injection points and linking exceptions with healing actions is defined by developers in XML configuration files. As in our model, this system allows attempting several healing actions in a serial way if the first attempts are not successful in removing the problem. But, this system does not provide a

way to automatically restore the system state after the occurrence of an exception. Furthermore, healing actions can be very specialized and, as in traditional exception handling, require the developers' expertise and implementation skills.

4.4. Summary

The exception handling model that we proposed in this chapter has a highly demanding set of objectives:

1. Our main objective is to influence developers to use exception handling mechanisms as a error recovery tool, and not only as a debugging aid;
2. At the same time, we want our model to be more tolerant to common exception handling programming bad practices - e.g., silencing exception, log-and-abort, code duplication, among others;
3. We want a model that is less intrusive to the programmer. A model that lets the programmer write business logic code without having to include large chunks of exception handling code just to deal with a potential exception;
4. The exception handling model must also contribute to increase code readability and comprehension. It must assure that the developer is aware of when exceptions can be raised and how they are handled, even if he or she did not write the exception handling code by himself;
5. Finally, the model has to provide for faster development and greater software reliability.

We believe that to achieve such objectives, our model as to take the greater part of the exception handling responsibilities away from the hands of the programmer. It must be able to prevent the programmer from writing as much of the exception handling code on an application as possible. In a sentence - "exception handling must become a platform issue, rather than the programmer responsibility".

In our model, exception handling is automatically performed by the runtime, as currently happens with memory allocation and garbage collectors.

Our proposal is supported by recent studies indicating that programmers neglect exception handling, that the quality and quantity of the code written for dealing with

errors is diminishing, and that the overall resilience to errors is condemned to suffer the effects of programming bad practices. Exception handling mechanisms are not being correctly used as an error recovery tool and the overall quality of exception handling code is very low. The source of the problem can be linked with design issues on the models themselves, with development strategies and weak requirements, and with the programmers lack of commitment with reliability issues. We believe that by freeing the programmer from writing error handling code, whenever viable, applications will be more robust and contain less latent bugs. Also, programmer productivity will be increased since much less code will have to be written, being that code non-trivial.

To provide developers with an automatic exception handling model, we have combined concepts from traditional exception handling mechanisms, backward error recovery systems (recovery blocks), and software transactional memory. Although, other techniques could be used to implement this model (e.g., AOP), our options provide automatic exception handling in a way that is completely transparent for the programmer.

The existence of a runtime level repository of recovery actions allows extensive reutilization of exception handling code. Being this code potentially heavily shared and used across applications means that more hidden bugs will be found and corrected, thus increasing the overall robustness of error handling and of the applications that rely on it. This approach also diminishes the mingling of business logic code and error handling code. The programmer, in general, does not have to think about error handling while thinking and concentrating on writing business code.

Implementation and Validation

This chapter presents a description of the implementation of the framework that supports the proposed exception model along with its evaluation.

We discuss the evaluation process and its results in order to assess the viability of automatically handling exceptions.

5.1. Introduction

It is impossible to advocate the qualities of a theoretical model without creating a *real-world-usable* implementation and running it through a set of well designed tests. Furthermore, when that model is about performing exception handling, thorough testing becomes even more important. Nonetheless, we are not recklessly assuming that only testing is, by any means, sufficient or complete. Our implementation of the Automatic Exception Handling (AEH) model, the tools that we have developed, and the tests that we performed can only take us so far as defending the model's viability and showing its main strengths/weaknesses. We are aware that the true value of the model will only be fully assessed when it is used "on the field", by thousands of developers, on projects with high reliability requirements. Our experiences, here discussed, are intended to show to the programming community at large that this model can represent a step forward for the development of more reliable software.

Our main objectives in the chapter are to:

1. Discuss a design for implementing the automatic exception handling model;
2. Assess the effectiveness of the model.

5.2. Framework implementation

The model described in the previous chapter was implemented for the Java 6 platform. It consists in three major components:

1. A modified Java compiler which makes `catch` blocks non-compulsory. The compiler is responsible for enforcing the semantics of the new exception handling mechanism are followed (e.g., non-transactional classes are not present in `try` blocks with no `catch`; exceptions are correctly re-raised; etc.). We used the open-source Jikes compiler [IBM2008], which was modified accordingly.
2. A custom-made Software Transactional Memory System implemented as a library. Although it would be possible to use other available STM libraries for Java, when we started the project there was a definite lack of such implementations. That led us to rollout our own system. It is a simple STM implementation based on object shadowing and supporting closed nested

transactions. It is aimed at supporting our proposed exception handling mechanism and not concurrency. Admittedly, in the future, we may replace it by a more mainstream STM library.

3. A Java system class loader which performs bytecode instrumentation at load time. That instrumentation supports the mechanism for proper invocation of the correct recovery blocks for each transactional try block, according to the application deployment configuration file, and the insertion of the appropriate code so that the transactional system is correctly invoked. It is also responsible for making sure that the correct application state is available for the recovery blocks so that they can, if needed, internally reconfigure the application.

Additionally, we have also implemented a small application that helps the programmer, or the person who deploys the application, to write the exception handling configuration file. It allows specifying for each exception type what recovery blocks should be executed and in what order. It also supports configuring specific recovery blocks to be executed by specific try locations. This application also allows specifying which state should be passed to each recovery block so that, if necessary, recovery blocks can internally reconfigure the application. Using this tool is not mandatory for our model since its main purpose is to free the programmer, as much as possible, from exception handling configuration-related tasks. Nevertheless, in practice, it is quite useful for deploying applications. This application will be described with greater detail later on this chapter but, for now, it is important to mention that it was implemented as an *Eclipse plug-in*. It identifies the blocks of instructions that are prone to raise exceptions and allows the developer to associate automatic recovery actions with the *offending* code, while ensuring that the corresponding deployment configuration file is updated on-the-fly.

The architecture of our system is illustrated in Figure 5.1. We have modified the Sun Java 6 JVM in order to implement our AEH model. In the figure, it is possible to observe the building blocks of the system organized accordingly to their run-time relationships.

The Java 6 JVM provides an ideal virtual run-time environment for applications using the AEH model. The JVM “sits” over the existent operating systems and uses a *just-in-time* compilation technique to allow the execution of the bytecode on the `.class` files of a program. In order to implement the AEH model, it was necessary to modify some of the system’s classes and libraries, providing new transactional implementations (*Transactional*

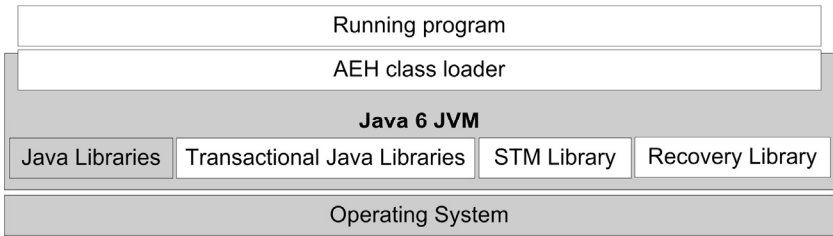


Figure 5.1 – AEH system architecture.

Java Libraries.) These libraries still work with the remaining default system libraries (left unmodified.)

Additionally, two new software packages were incorporated into the JVM: the *STM library*; and the *Recovery library*. The first package implements the types, interfaces and functionality necessary for creating the transaction environment for program execution, while the second implements the system recovery actions, and the methods for handler lookup and execution.

Figure 5.2 illustrates the process of loading and preparing a class for execution using the AEH model. Note that the applications do not know anything about transactions or recovery actions. The only modification introduced into the language and enforced by the modified compiler is possibility to use try blocks without catch or finally handlers. Thus, it is necessary to modify or introduce new code into the programs before execution in order to allow them to benefit from the AEH implementation. The Class Loader mechanism of the JVM allows us to perform such modifications on programs by intercepting the loading of new classes into the JVM, and allowing the modification of their code before they get executed for the first time. The *AEH Class Loader* has the tasks of: parsing the loading classes bytecode looking for pre-defined protected blocks of code; inserting the code that allows transactional execution of the classes' methods; inserting the code that allows the adequate binding of recovery actions to each exception occurrence; and inserting the code to execute the bound recovery actions when necessary. Basically, the AEH Class Loader is responsible for interconnecting the functionality provided by the STM library, the Recovery library, and the Transactional Java Library, with the code of the running programs. This functionality is provided by two *adapters*: the *TransactionClassAdapter* and the *RBClassAdapter*. The *TransactionClassAdapter* modifies the classes being loaded in order to incorporate the STM-related code and the *RBClassAdapter* modifies classes to append the recovery functionality. Both adapters use bytecode instrumentation to alter programs at load-time. Since we cannot intercept the

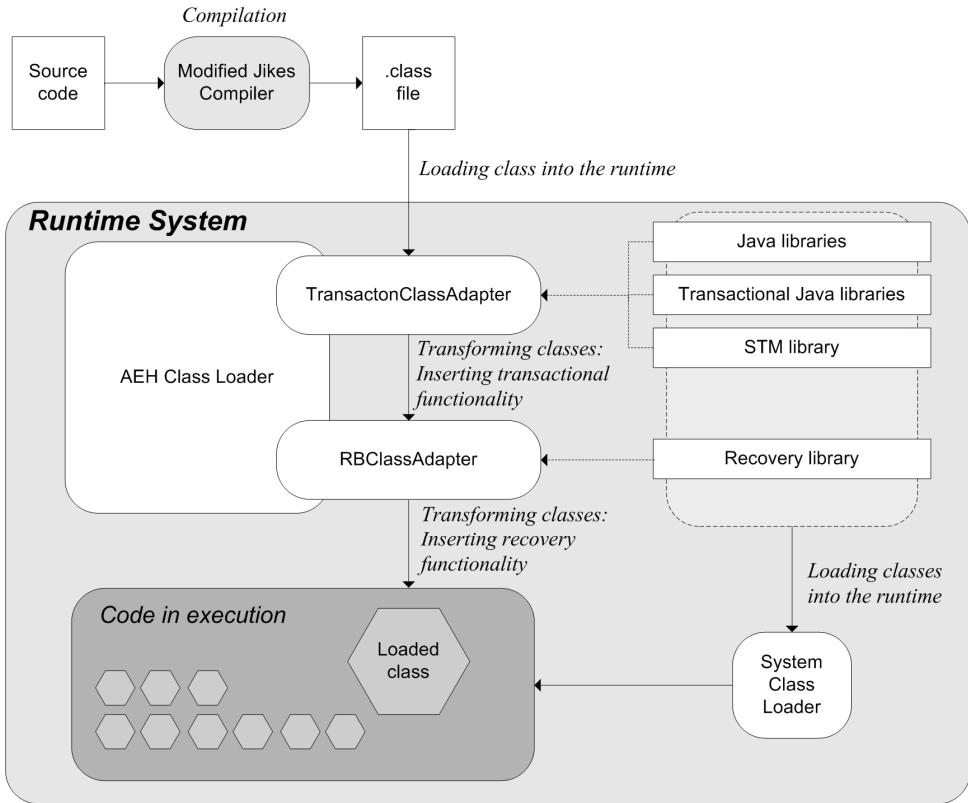


Figure 5.2 – Loading and running applications using the AEH.

loading of system classes, we needed to add some system specific classes by hand (Transactional Java Libraries package.)

The principles guiding the integration of our model into object-oriented software, using the AEH Class Loader, are similar to the basic concepts found on Aspect Oriented Programming methodologies. I.e., the functionality provided by our system is integrated into the targeted software in a crosscutting manner. On the other hand, our approach does not require that software developers must be aware of AOP concepts and language extensions since it is completely transparent and well integrated with object-oriented models.

```

public class Transaction {
    // transaction status: committed, aborted, running
    public int status;

    // thread id
    public long thread_id;

    // sequential transaction id
    public long trans_id;

    // reference to parent transaction in nested transactions
    public Transaction parent_trans = null;

    // local time at start
    public long start_time;

    // local time at end
    public long end_time;

    // list of objects touched by write operations in the
    // transaction
    public Set<ITransObject> wObjects;

    // list of objects touched by read operations in the
    // transaction
    public Set<ITransObject> rObjects;

    // Object to use for synchronization operations between
    // concurrent transactions when accessing the transactional
    // environment state
    public static Object trans_operations_lock = new Object();

    // List of transactions
    public static LinkedList<Transaction> transactionList =
        new LinkedList<Transaction>();

    // Constructor for a new transaction
    private Transaction() ...

    // Singleton method for creating and starting a new
    // transaction
    public static Transaction startTransaction() ...

    // The commit operation
    public boolean commit() throws Exception ...

    // The abort/rollback operation
    public void abort() ...

    // Remove a transaction from the list
    public void removeTransaction(Transaction t) ...

    // Get a sequential number representing time evolution
    public static long get_time() ...
}

```

Listing 5.1 - The Transaction class.

5.2.1. The STM Library

At the core of the STM library there is the `Transaction` class (Listing 5.1). This class implements the methods for creating, committing, and aborting a transaction. The transactional system contains a list of the transactions.

`Transaction` has a number of key operations and data structures. In particular, it holds a read-set (`robjects`) and write-set (`wobjects`) of objects that have either been read or touched during the transaction. When a new transaction is initiated, the system registers its parent thread identification, provides a new transaction identifier (sequential number), records the transaction `start_time` (sequential number representing time evolution), sets the transaction status to “running”, and adds the new `Transaction` object to the list of transactions. This list contains the transactions which are in execution, and which have been commit or aborted. Each time a transaction commits or aborts, the system performs a clean-up of the list eliminating terminated transactions. Note that not all objects can be removed from the list upon termination. In some situations, information about terminated transactions might still be needed to decide whether a running transaction will be able to commit or not.

Each transactional class in our system has to implement the interface shown on Listing 5.3. The methods in the interface are created on each class automatically using bytecode instrumentation techniques (see the AEH Class Loader section). These methods are essential for controlling object versions for each transaction. Since changes made inside the transactions are not visible to other concurrent transactions, it is necessary to maintain a list of object versions for each time a nested transaction is started. Objects on the system are organized on a double linked list. At the head of the list, we have the original object (the first instance to be created) and the following objects are copies made inside successive nested transactions. The first time a transaction touches an object with the intention of modifying it, the system makes a copy (the clone) and adds it to the instances list for that object. The copy can be based on the original object or on the active version on the parent transaction. When commit is performed, the system locates the original instance of the clone object being committed on the ending transaction and moves the changes of the clone onto the original.

```
public interface ITransObject {
    public ITransObject getNextTransObject();
    public ITransObject getPreviousTransObject();
    public Transaction getOwnerTrans();
    public void copy(Object destiny);
    public void updatePrevious();
    public void updateNext();
}
```

Listing 5.3 - The ITransObject interface.

```
public boolean commit() throws Exception
{
    boolean toCommit = true;
    // if the transaction is not running return
    if (this.status != RUNNING)
        return this.status == 0 ? true : false;
    this.end_time = Transaction.get_time();
    // access transaction environment constructs in mutual
    // exclusion
    synchronized (Transaction.trans_operations_lock) {
        // If there are no conflicts the transaction can commit
        if ((toCommit==canCommit())) {
            doFinalCommitTasks();
        }
        // else abort the transaction
        else {
            abort();
        }
    }
    return toCommit;
}
```

Listing 5.2 - The commit() method.

A `commit()` method (Listing 5.2) performs several verifications before allowing the changes made on ending transaction to be persisted. This method starts by setting the current `end_time` for the transaction (sequential number representing time evolution). Afterwards, the execution enters a mutual exclusive section of the code where it checks if there are any conflicting exceptions that can prevent the transaction from committing. Basically, if the read-set of a transaction intercepts with the write-set of another in the time intervals defined by the transaction `start_time` and `end_time` values.

```

private boolean canCommit() {
    boolean toCommit = true;
    // verify if a conflicting transaction exists
    for (Transaction t : this.transactionList) {
        // if another transaction committed, after this one had
        // already started, on a different thread
        // check for conflicts
        if (t.trans_id != this.trans_id
            && t.status == COMMITTED
            && t.end_time > this.start_time
            && t.thread_id != this.thread_id) {
            for (ITransObject o1 : t.wObjects) {
                for (ITransObject o2 : this.rObjects)
                    if (o1.id == o2.id) {
                        // if this transaction reads from an object that
                        // was modified on the previously committed
                        // transaction it should not be allowed to commit
                        // because commit occurred after the transaction
                        // started and the new value was not considered
                        toCommit = false;
                        break;
                    }
            }
            for (ITransObject o3 : this.wObjects)
                if (o1.id == o3.id) {
                    // the same as above but for write operations
                    toCommit = false;
                    break;
                }
            if (!toCommit)
                break;
        }
    }
    for (ITransObject o1 : t.rObjects) {
        for (ITransObject o2 : this.wObjects)
            if (o1.id == o2.id) {
                // if this transaction modified an object
                // that was read on the committed transaction
                // it should not commit because the new value
                // was not accessible for the previously committed
                // transaction
                toCommit = false;
                break;
            }
        if (!toCommit)
            break;
    }
}
return toCommit;
}

```

Listing 5.4 - The canCommit() method.

The method `canCommit()`, shown on Listing 5.4, is responsible for checking for possible conflicts while performing `commit()`. If this method returns `false`, the transaction is aborted, otherwise the changes on each object that was modified during the transaction are persisted (by execution of `doFinalCommitTasks()`). Each time this method executes, it verifies if there was any other transaction being committed after the current transaction was initiated and, if that was the case, ensures that there are no conflicts. This happens

```

private void doFinalCommitTasks() {
    ITransObject original = null;
    ITransObject clone = null;
    // For each object modified inside the transaction
    // Make the changes persistent on the original instance
    for (ITransObject o : this.wObjects) {
        if (this.parent_trans != null) {
            original = o;
            while (true) {
                original = original.getNextTransObject();
                if (original == null)
                    break;
                if (original.getOwnerTrans().trans_id ==
                    this.parent_trans.trans_id)
                    break;
            }
        }
        if (original == null)
            original = o;
        clone = o;
        while (true)
        {
            clone = clone.getNextTransObject();
            if (clone == null)
                throw new Exception("Clone not found");
            if (clone.getOwnerTrans().trans_id == this.trans_id)
                break;
        }
        clone.copy(original);
        clone.updatePrevious();
        if (clone.getNextTransObject() != null)
            clone.updateNext();
        clone = null;
        original = null;
    }
    this.status = COMMITTED;
}

```

Listing 5.5 - The doFinalCommitTasks() method.

because changes made inside the transactions are only visible to other transactions after commit. The canCommit() method will only return true if objects modified on a transaction that was committed after the current one had already been initiated were not accessed (read or write) on the current transaction.

The doFinalCommitTasks() method (Listing 5.5) uses the information (and methods) on each transactional object to locate the original instance of the clone object being committed on the ending transaction and to make persistent the changes of the clone onto the original.

5.2.2. The AEH Class Loader

A custom class loader is used to intercept the loading of classes into the runtime system (JVM). By doing so, we are able to instrument the original code of the classes and provide the JVM with new transaction-enabled implementations of those classes.

To perform the instrumentation of metadata and the bytecode on the `.class` files, we used the ASM library [Bruneton2002]. Although, there are other options in terms of Java bytecode instrumentation libraries (e.g., [Dahm1999,Chiba2000]), ASM, due to its design and operating model, provides excellent performance and a low memory footprint, being these qualities essential to our system.

We used the ASM library to create two *class adapters*, which are called by the `AEHClassLoader` to perform the actual instrumentations:

1. `TransactionClassAdapter` – makes the loading classes *transaction-aware*;
2. `RBCClassAdapter` – inserts the code to invoke the recovery actions when exceptions are raised inside `try` blocks.

The Transactions Class Adapter

The `TransactionClassAdapter` is responsible, among other things, to assure that the loaded classes implement the `ITransObject` interface correctly. The `TransactionClassAdapter` class extends the `org.objectweb.asm.ClassAdapter` class and implements the `org.objectweb.asm.Opcode` interface. The first reference is the base class for all adapters implemented for the ASM, and the second an interface used for allowing the access to the bytecode instrumentation functionality inside ASM. The complete list of the methods available on this class is shown on Listing 5.6.

The ASM library implements the Visitor design pattern [Gamma1995]. This allows our modifications to propagate through an application's code in a systematic way. The visitor pattern provides the means to instrument (visit) each component of a program (classes, fields, methods, and constructors) in a different way.

When a class is first loaded by the class loader, ASM adds a new interface to it (`ITransObject`). This interface allows the runtime system to duplicate objects as needed by the transactional system. It also allows the transactional system to navigate through the list of shadow copies created when a new transaction is started, committed or aborted, adding, removing or updating the corresponding copies as needed. Finally, this interface allows the runtime system to gather context information about parent transactions when nested transactions are taking place.

```

public class TransactionClassAdapter extends ClassAdapter
implements Opcodes {

    // The constructor
    public TransactionClassAdapter(ClassVisitor cv, ClassNode
    cn);

    /* Methods used by the ASM ClassAdapter to implement the
    * Visitor design pattern and apply transformations to
    * the applications' code
    */

    // The class visitor
    public void visit(int version, int access, String name,
    String signature, String superName, String[] interfaces);

    // The method visitor
    public MethodVisitor visitMethod(int access, String name,
    String desc, String signature, String[] exceptions);

    // The fields visitor
    public FieldVisitor visitField(int access, String name,
    String desc, String signature, Object value);

    // The instrumentation finisher
    public void visitEnd();

    /* Private methods used to implement the new components
    * into the existing classes
    */
    // If the class has static fields, we build a class wrapper
    // for dealing with them
    private void buildStaticClass();

    // Add the fields and the fields' accessor methods required
    // by the transactional functionality and the
    // getOwnerTrans() method
    private void addFields();

    // Add the getNextTransObject() and getPreviousTransObject()
    private void addGettersAndSetters();

    // Add the copy() method, used by the transactional system
    // to make the values of the clone objects permanent on
    // commit
    private void addCopyMethod();

    // Add the updateNext() and updatePrevious() methods
    private void addTransObjectUpdaters();

    // Add cloning method
    private void addCloneMethod();
}

```

Listing 5.6 - The TransactionClassAdapter() methods.

A TransactionMethodAdapter is used for modifying the bytecode of each method on the loaded classes. Just adding mechanisms for creating and updating objects by using code

instrumentation is not enough. It is essential to ensure that the code of the modified classes uses the fields and variables of the correct objects within a transaction. This means that the bytecode of the classes has to be modified or it would access the original object references. Thus, the following modifications are made:

1. Replace the `PUTFIELD`, `GETFIELD`, `PUTSTATIC`, `GETSTATIC` instructions by calls to new field access methods;
2. Replace the `ILOAD`, `FLOAD`, `ALOAD`, `LLOAD`, `DLOAD`, `ISTORE`, `FSTORE`, `ASTORE`, `LSTORE`, `DSTORE`, and `RET` instructions by call to the local variables access methods;
3. Locate the `retry` instructions (actually, for simplicity we opted for using a method call instead of special instruction) on the code and replace them by jumps to beginning of the `try` code block to be retried [Cabral2009];
4. Insert the calls to `startTransaction()`, `commit()`, and `abort()` methods at start of `try` blocks, at the start of `finally` blocks (in some cases it is necessary to append a new `finally` block where one does not exists), and exception handlers respectively.

The Recovery Actions Adapter

The `RBClassAdapter` is instantiated (for each class being loaded) only after the `TransactionClassAdapter` finishes its execution. This adapter is responsible for including the automatic recovery code into the program. Basically, the adapter uses the `RBMethodAdapter` to modify the target bytecode for each method in a class.

`AEHClassLoader`, and `RBMethodAdapter` in particular, use the information on configuration files to control the process of inserting the recovery code into applications at load-time. The adapter uses that knowledge base to know which recovery actions are to be associated with a particular exception type, class, method or protected block. The adapter detects any transactional `try` blocks present in the code and, with this information and the configuration data, either inserts the functionality of the handler directly on the target code if the code is small or inserts a selection mechanism controlling the invocation of the adequate recovery methods. These calls will require establishing a relation between the failing component's local variables and the recovery methods' parameters.

```

import javax.jms.*;

public class JMSExceptionRecovery {

    public static void recovery_01() {
        retry;
    }

    public static void recovery_02() {
        Thread.sleep(1000);
        retry;
    }

    public static void recovery_03(
        ConnectionFactory connectionFactory,
        Connection connection, Session session, Queue queue,
        String queueName) throws JMSException
    {
        connectionFactory =
            SampleUtilities.getConnectionFactory();
        connection = connectionFactory.createConnection();
        session = connection.createSession(false,
            Session.AUTO_ACKNOWLEDGE);
        queue = SampleUtilities.getQueue(queueName, session);
        retry;
    }

    public static void recovery_04(ConnectionFactory
        connectionFactory,
        Connection connection, Session session, Queue queue)
        throws JMSException
    {
        Properties properties = new Properties();
        properties.load(new
            FileInputStream("execution.properties"));
        connectionFactory =
            SampleUtilities.getConnectionFactory();
        connection = connectionFactory.createConnection();
        session = connection.createSession(false,
            Session.AUTO_ACKNOWLEDGE);
        queue = SampleUtilities.getQueue(
            properties.getProperty("queueNameAlternative"),
            session);
        retry;
    }

    public static void recovery_05() {
        System.out.println("The system was not able to recover
            from the exception automatically." +
            "\nDo you wish to (T)ry again, (R)ethrow the exception,
            (A)abort the program execution?");
        ...
    }
}

```

Listing 5.7 – Recovery actions for the JMSException class.

In our system, we defined recovery actions as methods, belonging to special classes. These classes are part of the runtime platform. Listing 5.7 shows an example of one of these classes. This particular class was used for testing the platform, as we will describe on the next section. In the virtual machine, default recovery actions can be defined inside *sister*


```

<?xml version="1.0" encoding="UTF-8"?>
<recovery_bindings>
  ...
  <entry exception="JMSEException"
        recovery_class="JMSEExceptionRecovery">
    <packages />
    <classes />
    <methods>
      <method>
        <className>SenderToQueue</className>
        <methodName>main</methodName>
        <desc>([Ljava/lang/String;)V</desc>
        <recovery_method name="recovery_03">
          <parameter_var name="connection">
            connection
          </parameter_var>
          ...
        </recovery_method>
      </method>
    </methods>
  </entry>
</recovery_bindings >

```

Listing 5.8 - Configuration file example.

classes of the existent exception classes. Meaning that, for each exception type, a set of recovery actions can be defined inside a new class. To differentiate these sister classes from normal classes, they must be marked. And, although we can find many different forms of inserting a distinguishing factor in classes, such as Java Annotations, .NET Custom Attributes, or simple object-oriented inheritance, we believe that the most traditional approach of making these classes implement a basic interface is the most benefic. Not only interfaces are useful to define the core functionality that these classes must offer, they are also an excellent way for creating compile-time checks and verifications on the safety and correctness of code.

As the class name suggests, this recovery class is generally associated (at platform level) with the `JMSEException` type. Nevertheless, other bindings can be set. The system configuration file, formatted as an XML file, can be freely modified and adapted to the requirements of each application. It is also possible to provide new configuration files for different programs.

Listing 5.8 shows an extract of a configuration file used for testing purposes. In this example, we are binding the `JMSEExceptionRecovery` class to the occurrences of the `JMSEException` on the `main()` method of `SenderToQueue` class.

Another important aspect that can emerge from the example is the overall complexity of configuration files, and how that complexity can become overwhelming for larger

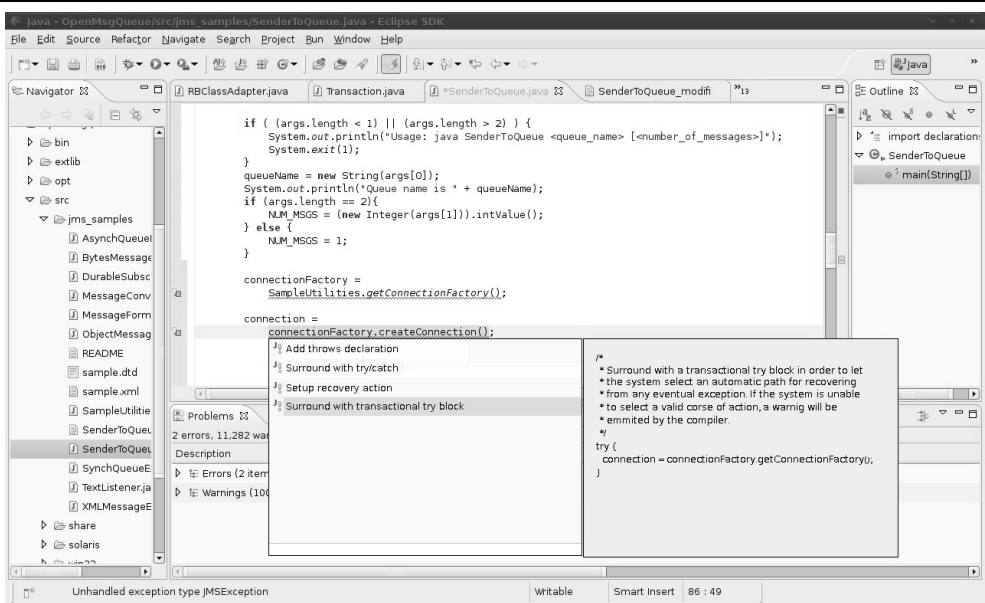


Figure 5.3 – Eclipse plug-in.

programs. For instance, setting up the binding between the method’s local variables/fields and the recovery methods’ parameters is relatively cumbersome. Fortunately, many of the elements on the file can be left blank most times, because the system is able to automatically generate most of that information. On more complex situations, the programmer can be aided by visual tools that make the generation of configuration file semi-automatic.

Consider, for example, that when using an IDE, such as Eclipse, the editor alerts the programmer of any unhandled exceptions, and gives him different options on how to deal with those exceptions (Figure 5.3). In the figure, our Eclipse plug-in is alerting the programmer for an unhandled `JMSEException` on the code and proposes four different ways of dealing with the exception. The first two options are inherited from the traditional Java exception handling model, the last two are new:

1. *Setup recovery action* – This option will automatically surround the code line with a transactional `try` block and open a configuration window that will help the programmer setup the recovery action, methods and parameters for dealing with the exception. We have developed the application on Figure 5.4 to help us build configuration files.

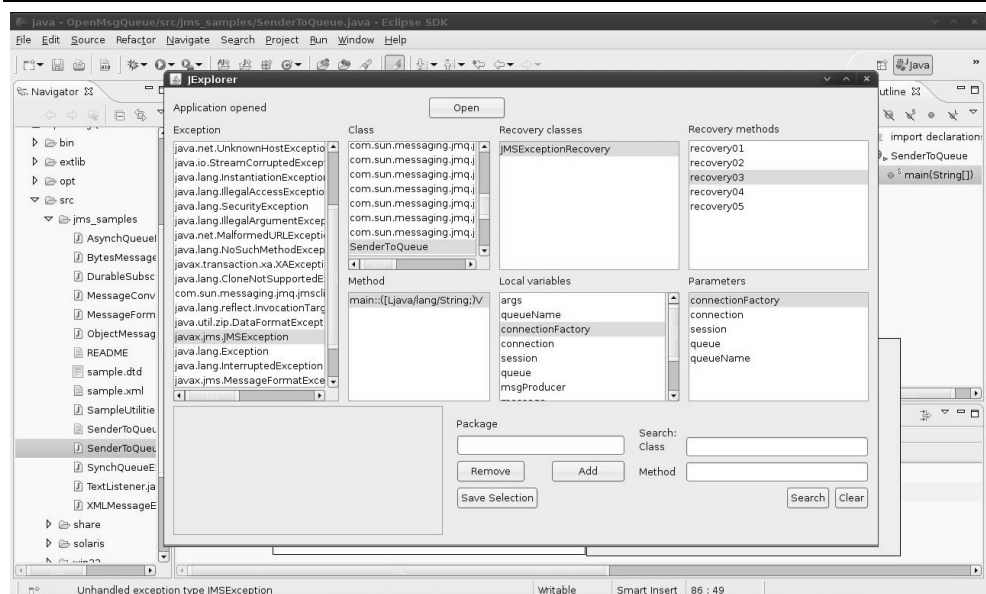


Figure 5.4 – Configuration interface.

2. *Surround with transactional try block* - This option will automatically surround the code line with a transactional try block and let all configuration to be performed automatically by the run-time system. If the system is unable to setup the handling actions alone, it will alert the developer for that fact during compilation or later execution.

5.3. Validation and Testing

To evaluate the effectiveness of the approach, we used 11 different programs. These consisted in the applications that come with the Sun's Community Version of its Java System Message Queue (MQ) framework on the GlassFish framework [Sun2008], implementing the JMS standard. We have chosen this system because it is a widely used platform and, at the same time, the test applications are not so complex, allowing focused experiments to be made.

Since the main goal of the proposed approach is to increase application resilience and, at the same time, decrease the amount of exception handling code written by the programmer (whenever possible). Four vectors of evaluation were used:

1. Amount of source code written by the programmer;

2. Effect on the application's resilience;
3. Performance penalty imposed by the exception runtime;
4. Viability of providing general recovery actions.

5.3.1. Source Code

Using a parser (the same we used for parsing Java code on Chapter 3), we analyzed all the exception handling code both from the JMS server and the 11 applications. Exception handling code corresponds to 10% of the overall code, which consists in 21.666 lines of code. We also verified that in many cases the exception handling code inside the JMS server transforms raised exceptions into a generic exception type: `JMSEException`. This exception is then re-thrown.

Overall to the test applications and server, 2.170 lines correspond to code that handle `JMSEException`, accounting for 1% of all code, being the most common exception used. Being the `JMSEException` so central, it was a prime candidate for being handled automatically.

We analyzed the documentation associated to the `JMSEException`, its semantics, and how it is actually being used in the code base. Twenty five causes were identified as a reason for such exception to occur. These are shown on Table 5.1.

Many of these problems can actually be solved (or be tried to solve) using a small set of benign recovery actions:

1. Immediately re-execute the operation;
2. Pause for a predetermined time and re-execute again;
3. Reinitialize the connection to the JMS broker and re-execute;
4. Modify the connection properties to use a different broker, if configured, and re-execute;
5. Notify the user of the problem (detail on the exception cause), allowing him to manually correct the problem before re-executing.

If all actions fail, the exception is simply re-thrown.

Table 5.1 – Causes for `JMSEException` to be raised.

JMSEException's causes
Error validating the host:port string format
Error reading properties
Invalid object class when wrapping a standard JMS ConnectionFactory administered object
Invalid JMSSelector
Null client id in connection object
Invalid client id
Invalid client id in Connection
Invalid connection on the current session
Invalid URL string in connection object
Connection is closed
Connection object is null
Connection not started
There is an active consumer on destination when doing delete
There is an active consumer on destination when doing unsubscribe
Unsupported operations
Invalid Queue object
Invalid Topic object
Invalid Queue or topic name
Invalid message status (properties)
Invalid delivery mode
Invalid priority
Invalid time to live
Decompression error
Invalid acknowledge mode

The 11 test applications were re-written using the proposed exception model. This consisted mostly in removing the catch blocks that deal with `JMSEException`, becoming that handling automatic. It allowed us to remove 143 exception handling blocks, representing more than 30% of all exception handling source code. This is significant since it shows that, when it is possible to use automatic exception handling, the programmer will be able to write much more concise code¹.

¹ Note that if the programmer wants to manually treat an exception, that is always possible since catch blocks take precedence over automatic handling.

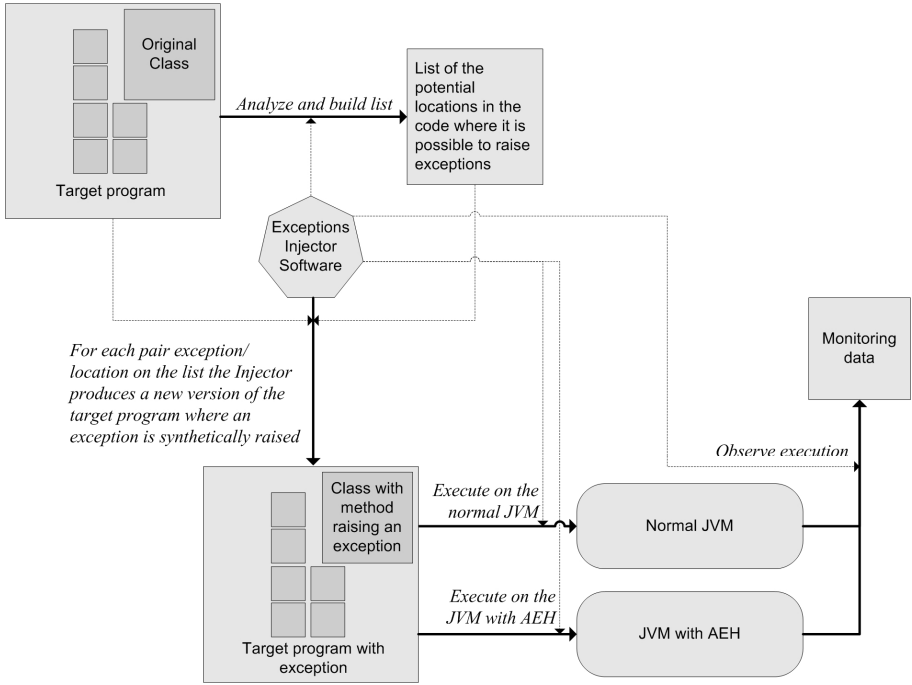


Figure 5.5 - Testing framework.

5.3.2. Resilience

In the following subsections we will describe the usage of the Automatic Exception Handling model on two distinct scenarios and discuss its influence on the overall systems resilience to errors.

Use case: Java Messaging System

For testing the robustness of the code written using the automatic exception handling model relatively to the original code, we built a fault injector that systematically raises exceptions inside of try blocks, monitoring the subsequent behavior of the applications.

The rationale is simple: if there is a try block for handing an exception, if the application is robust, then when such exception occurs the application should behave in a sensible way. It should not produce an incorrect result nor crash.

Figure 5.5 illustrates the overall functionality of the exception injection software and the testing framework. The injector software creates a list of all possible locations in the target programs where an exception can be raised. For each exception and location identified on

this list, the injector creates a new version of the target software where the exception is raised at the desired location. This *faulting* version is then executed on a *normal* JVM and on a JVM implementing the AEH mechanism. The output of each execution is then registered for posterior analysis.

In our system, after raising an exception, its effect on the program is observed from the injector software, which records the outcome in a database. The effects are classified in one of three different categories:

1. **Abort:** the application aborts its execution;
2. **Successful:** the application does not abort and its output is the same as an execution with no errors;
3. **Incorrect:** the application does not abort but its output is different from an execution with no errors.

Table 5.2 shows the overall results of injecting 216 `JMSException` across the applications.

Table 5.2 - Exception Injection Results (all apps.)

	# Exceptions	Successful	Abort	Incorrect
Unmodified Applications	100% (216)	20% (44)	79% (171)	1% (1)
Automatic Exception Handling	100% (216)	100% (216)	0% (0)	0% (0)

As it can be seen, on the unmodified applications, 20% of the raised exceptions did not produce any different observable result compared with a normal execution (“golden run”). Note that this does not mean that the applications did what they should. It only means that their output was identical. 79% of the injected exceptions led to the applications crashing. In 1% of the cases, the output of the applications was different from the one that would be obtained on a non-erroneous execution. When we performed the same experiments using automatic exception handling, in all cases, the simple five recovery mechanisms that were implemented lead to “Successful” executions. Again, this does not mean that the internal state of the applications was not corrupted; it only means that no external effects were seen. Even so, the results are somewhat impressive: by using simple recovery strategies there are less 79% application crashes. And, at the same time, 699 lines of code are

eliminated, corresponding to 143 handlers. This is code that the programmer will not have to write. With less code we are achieving higher resilience. Of course, the original programmers could have embedded the five recovery blocks that we designed in the original catch handlers. But, the facts are: a) they did not; b) the overall code would have been much more complicated and long. These findings are in line with the results presented in [Cabral2007], where 32 highly used applications were examined, and with [Shah2008a].

As we mentioned before, the fact that applications did not produce different results does not mean that they were doing what they should. This is especially true on our set of applications since they mostly send and receive JMS messages. For investigating this issue, three applications were selected where it was easy to monitor if sent messages actually reached the destination and their contents were uncorrupted. Again, exceptions were raised, and both the execution of the source application and the contents of messages at the receiving application were examined. The effects of the exceptions were classified in the following categories:

1. **Abort - Correct Delivery:** the application aborts but the message it tried to send reached its destination correctly;
2. **Abort - Incorrect Delivery:** the application aborts and the message it tried to send either did not reach its destination or its contents were corrupted;
3. **Successful:** the application does not abort and the message reached the destination correctly.

Notice that the “Incorrect” category does not exist in this case since the sending applications do not produce output. The next table (Table 5.3) summarizes the results.

Table 5.3 – Results with content checking (3 apps.)

	# Exceptions	Successful	Abort - Correct Delivery	Abort - Incorrect Delivery
Unmodified Applications	100% (25)	12% (3)	24% (6)	64% (16)
Automatic Exception Handling	100% (25)	100% (25)	0% (0)	0% (0)

The most interesting result from Table 5.3 is that in 24% of the cases, although the applications aborted, they were able to send their messages correctly, which was their primary goal. This suggests that with little effort these applications could be much more resilient, which is actually verified by looking at the results of the automatic exception handling.

Overall, although our experiments were limited in scope, the results appear to indicate that the automatically exception handling approach can have a dramatic impact on the robustness of applications.

Performance Analysis

Our approach to exception handling has some performance overhead. Since each block is transactional, in our implementation, it implies to create a shadow copy of each object that is updated inside a try block. Even so, the impact is not as high as one might expect. The reasons are: 1) try blocks are normally relatively short and are not deeply nested. This means that the number of objects that are touched inside of a block is typically small. 2) try blocks are not normally re-executed. This means that when a block commits, most of the work consists in a small number of checks and on substituting the original object references by the updated copies. Comparing with “normal” STM systems, our case is similar to the situation where the code executes without contention from other threads and commits almost every time. This means it can be heavily optimized.

For assessing performance we used the test applications which could be run in a loop (6 of the applications), sending/receiving or processing messages. Each application was run for 100 loops. As it can be seen on the graph (Figure 5.6), the overall performance impact is negligible.

Use case: Hipergate server

Despite the promising results, we decided to evaluate the reliability of our model in more realistic scenario. For doing so, we selected an application that uses a database server for persisting data - the *Hipergate CRM Groupware tool* [Knowgate2006] and prepared a test where the connection to the database was lost, during execution, in order to observe the system behavior while using automatic recovery.

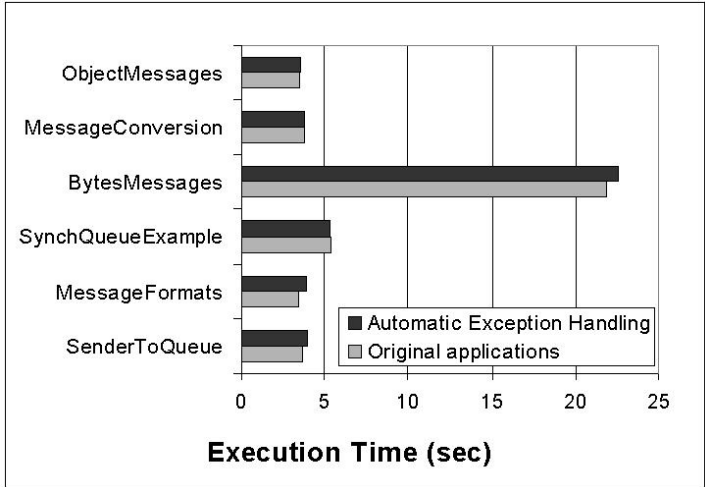


Figure 5.6 – Analysis of the executions times.

Hipergate is a CRM dynamic web application that runs on top of the Tomcat [Apache2009a] server and uses a MySQL [Sun2009] database to persist data. After installing, configuring, and inserting the initial data into the application, we used Jakarta JMeter [Apache2009d] to simulate a workload of 10 accesses per second to Hipergate during a 10 minutes run with a 30 second timeout for each request. What this means is that a user would never wait more than 30 seconds for a web page, independently of the cause of the delay. Furthermore, to evaluate the system recovery ability using automatic exception handling, we planned to shutdown the database for 30 seconds after the first 5 minutes of the test. On this scenario, the Mean-Time-Between-Failures (MTBF) is 9,5 minutes and the Mean-Time-To-Repair (MTTR) is 0,5 minutes. Thus the system availability is 95%. This is shown in Figure 5.7. In broader terms, if we do not use automatic exception handling, the system will not be able to respond to the caller with a valid page (code 200 in the HTML protocol) 5% of time, i.e., in 30 seconds.

We prepared our application to automatically handle the `SQLException` type. For testing purposes we consider that the only cause for the occurrence of this exception is the use of a lost or previously closed database connection. The system recovery code reacts to the exception by executing 3 retry attempts separated by 5 seconds each. Each time the application retries it also attempts to open a valid database connection. This means that if the connection is re-established during this 15 seconds interval, the application will be able to respond with a valid result. If not, the exception is propagated in a normal way. Going back to the scenario previously described, and looking at the 30 seconds interval when the

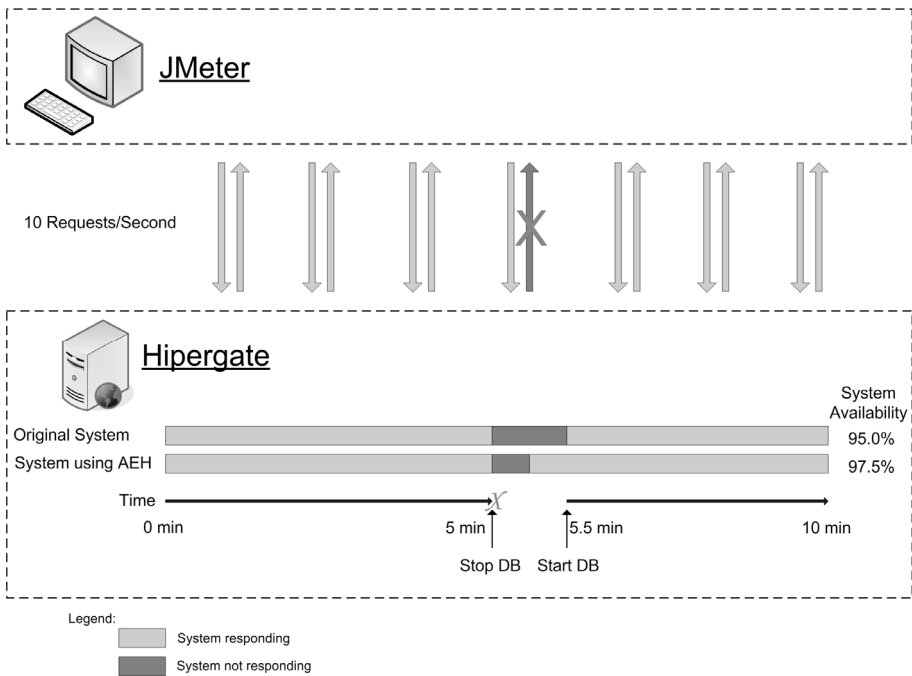


Figure 5.7 – Description of the experience.

database is down, we can expect that the requests made during the first 15 seconds will not be fulfilled but the ones done on the second half will be able to obtain a valid response. This would represent 2,5% less errors and an error rate of 2,5%. The system availability should increase to 97,5%. To validate our predictions we used two separate machines, a server, where Hipergate, Tomcat 6 and a MySQL were running, and a client machine where we executed the JMeter workload and saved the results. Figure 5.7 illustrates the experience. We performed 2 types of runs always using the workload configuration but changed the server settings: (a) in the first run we did not shutdown the database and obtained a 0% error rate; (b) on the second run we shut down the database and obtained an error rate of 2,62% (Figure 5.8). The value of the standard deviation shown on Figure 5.8 is very high because: a) when the database is running, response times are usually very quick (less than 1 second); but when the database is down, b) responses will *timeout* after 30 seconds; or c) require up to 15 seconds to be fulfilled.

These tests have shown that the system availability increased up to 97,38% when using automatic exception handling. Nonetheless, as it can be observed in Table 5.4, the error rate is slightly superior to the predicted value. There are mainly two reasons that can contribute this outcome: (i) the throughput is not exactly 10 requests/second but 9,9

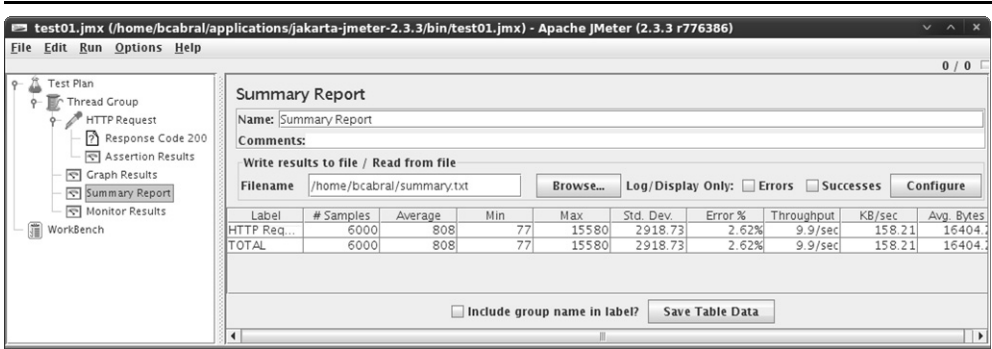


Figure 5.8 – JMeter workload run summary.

requests/second; and, (ii) we cannot control in detail the down time of the database, we can only guarantee that between the issuing of the stop and start commands there is a 30 seconds interval, we do not know how much time the database takes to start and stop its service (our experiments show that this interval varies between 1 and 3 seconds.)

Table 5.4 – System availability, MTTR, MTBF and error rate.

Description	System Availability	MTTR	MTBF	Error Rate
System run without interruption of the DB service	100%	0.00 min	N/A	0.00%
System run experiencing the interruption of the DB service	95%	0.50 min	9.50 min	5.00%
System run using automatic exception handling while experiencing the interruption of the DB service	97.38%	0.26 min	9.74 min	2.62%

It is perceptible that our model promotes an effective increase on the overall reliability of the targeted system with a very simple recovery strategy. If we consider that this increase is obtained with the same amount of source code or even less, the drop of nearly 50% on the error rate is a remarkable result.

5.3.3. Recovery Actions

Recovery actions are a core component of our model. Thus, it is essential to measure to what extent will system designers be able to ship sets of automatic and benign recovery

actions with their products. Though, only real world experience will give us an undoubtedly perspective on the subject, the model evaluation would not be complete without tackling this issue.

Our experiments, while simple are also very promising. We decided to select a large number of the Java's platform system exceptions and provide automatic handlers for them. By doing so, we intended to show that it is possible to define automatic recovery actions for a large number of exception types. Secondly, we tried to assess the impact that those automatic handling policies would have on the reduction of length of programs' source code.

We decided to tackle all the exceptions types that directly or indirectly descend from the `java.io.IOException` class on the Java 6 API. Our choices are summarized on Table 5.5.

Table 5.5 – Recovery actions for Java's `IOException` class tree.

Exception Class	Recovery Actions
<code>ChangedCharSetException</code>	Obtain the <code>charset</code> from the exception object and replace the <code>charset</code> on the <code>reader/streamreader</code> object
<code>MalformedInputException</code>	<code>getInputLength()</code> can be called to determine the length of the bad input and throw it out or fix it
<code>UnmappableCharacterException</code>	<code>getInputLength()</code> can be called to determine the length of the bad input and throw it out or fix it
<code>ClosedChannelException</code>	Re-open the channel; Re-open the channel with privileges for the desired operation
<code>AsynchronousCloseException</code>	Re-open the channel
<code>FileLockInterruptedException</code>	Retry
<code>HttpRetryException</code>	Disable streaming mode and retry
<code>InterruptedIOException</code>	Resume the transfer from the interruption point in the data (<code>bytesTransferred</code>)
<code>SocketTimeoutException</code>	Retry; Re-open socket and retry
<code>InvalidClassException</code>	Try to load the class from a different location; Notify the user of the problem (detail on the exception cause), allowing him to manually correct the problem before retrying or aborting
<code>SocketSecurityException</code>	*Obsolete
<code>UnknownHostException</code>	Connect to a different host and retry; Notify the user of the problem (detail on the exception cause), allowing him to manually correct the problem before retrying or aborting

Exception Class	Recovery Actions
ConnectException ConnectIOException ConnectException NoRouteToHostException PortUnreachableException	Retry ; Retry after pausing for predefined time period; Connect to a different host and retry; Notify the user of the problem (detail on the exception cause), allowing him to manually correct the problem before retrying or aborting
BindException	Retry; Retry after pausing for predefined time period; Bind another alternative port; Notify the user of the problem (detail on the exception cause), allowing him to manually correct the problem before retrying or aborting
SSLHandshakeException	Retry; Modify connection settings, reconnect and retry; Notify the user of the problem (detail on the exception cause), allowing him to manually correct the problem before retrying or aborting
SyncFailedException	Retry; Notify the user of the problem (detail on the exception cause), allowing him to manually correct the problem before retrying or aborting
UnknownHostException	Modify or replace the host name; Notify the user of the problem (detail on the exception cause), allowing him to manually correct the problem before retrying or aborting
UnsupportedDataTypeException	Use alternative data type and retry; Notify the user of the problem (detail on the exception cause), allowing him to manually correct the problem before retrying or aborting
UnsupportedEncodingException	Use alternative character encoding; Notify the user of the problem (detail on the exception cause), allowing him to manually correct the problem before retrying or aborting
UTFDataFormatException ZipException JarException UnknownServiceException FileNotFoundException FileException IOException InvalidPropertiesFormatException JMXProviderException JMXServerErrorException MalformedURLException InvalidObjectException OptionalDataException ProtocolException AccessException AuthenticationException SSLKeyException SSLPeerUnverifiedException	Notify the user of the problem (detail on the exception cause), allowing him to manually correct the problem before retrying or aborting

Exception Class	Recovery Actions
SSLProtocolException SSLException SocketException ActivityCompletedException ServerError ServerException ServerRuntimeException StubNotFoundException TransactionRequiredException CharConversionException EOFException IIOInvalidTreeException ObjectStreamException RemoteException UnexpectedException	Not possible to handle automatically
NotActiveException NotSerializableException StreamCorruptedException WriteAbortedException ActivateFailedException ActivityRequiredException ExportException InvalidActivityException InvalidTransactionException MarshalException NoSuchObjectException TransactionRolledbackException UnmarshalException SaslException	Not possible to handle automatically due to insufficient information about internal details

From the 70 exception classes under analysis, we were able to define benign automatic recovery actions for 41 (60%). We decided not to handle 15 (21%) exception types automatically. And, we were not able to decide which would be the preferable course of action on 14 (19%) situations. This was mostly due to the lack of information on the documentation about the internal details associated with raising and handling these exception types.

Considering that we were able to automatically handle more than half the IO-related exceptions on a system, these results look quite promising. Consequently, we believe that the robustness, development time, and simplicity of source code would be positively influenced by these actions. Of course, our recovery actions do not guarantee that exception occurrences will always be successfully handled. The outcome of the execution of each recovery block and, subsequently, the re-execution of the protected regions will always depend of the runtime state and environment when the exception occurs. Nevertheless, if we recall what we have already learned about nowadays exception handling code and practices, our automatic system provides higher guarantees of success.

As we have already seen, providing recovery code for occurring exceptions allows the programmer not to write exception handling code for every exception on his programs. Thus, the overall size of programs’ source code will decrease. To assess the impact of the recovery actions that we propose on the size of the source code of existing applications, we selected a set of applications with high IO demands and accounted for the amount of code that would be unnecessary if the referred exceptions were automatically handled. The results are listed on Table 5.6.

Table 5.6 – Applications source code decrease analysis.

Applications	# LOC	# EH LOC	# EH LOC (IOException family)	# EH LOC (automatically handled types)	# Catch Blocks	# Catch Blocks (IOException family)	#Catch Blocks (automatically handled types)	Maximum-Minimum decrease on EH code size
j-ftp	15359	690	148	10	244	48	4	21%-2%
Lucene	56362	1087	447	27	434	175	15	41%-3%
Tapestry	65984	1259	72	22	490	34	9	6%-2%
Columba	89325	2291	597	193	787	184	57	26%-8%
Limewire	342393	6853	2596	471	3327	1216	258	38%-7%

Legend: (a) **Applications**, name of the programs under analysis; (b) **#LOC**, total number of lines of code on each program; (c) **#EH LOC**, number of lines of code dedicated to exception handling on each program; (d) **#EH LOC (IOException family)**, number of lines of code dedicated to detecting and handling exceptions on the `IOException` class hierarchy; (e) **#EH LOC (automatically handled types)**, number of lines of code dedicated to detecting and handling the exception types chosen for automatic recovery and identified on Table 5.5; (f) **#Catch Blocks**, total number of catch blocks on each program; (g) **#Catch Blocks (IOException family)**, number of catch blocks dedicated to handling exceptions on the `IOException` class hierarchy; (h) **#Catch Blocks (automatically handled types)**, number of catch blocks dedicated to handling exceptions of the types chosen for automatic recovery and identified on Table 5.5; (i) **Maximum-Minimum decrease on EH code size**, the minimum decrease corresponds to the elimination of all handling code for the exception types chosen for automatic recovery and identified on Table 5.5, the maximum decrease corresponds to the elimination of all code dedicated to handling all the exceptions on the `IOException` family.

When analyzing the results on the table, we have to consider several aspects associated with design options taken on these programs. First, these applications were not written with our exception model in mind. Moreover, most of these applications rarely use exception handling code for recovery. Thus, the amount of code dedicated to promoting reliability on these programs is very small. This fact is visible when comparing the values of the overall number of lines of code in each program with the total number of lines of code dedicated to exception handling. In average, only 2,6% of the code in these applications, with a standard deviation of 1%, is dedicated to exception handling. Furthermore, most of code in exception handlers is used to *silence exceptions, log exceptions*

or *log and abort the program*. Since the AEH model promotes recovery, it is no surprise that it shows a higher influence on the resilience of existing applications, designed for using the traditional exception models, than on the size of their source code. On the other hand, if we were to incorporate the code of all recovery blocks into the exception handlers on these applications to make them more robust, the amount of code dedicated to exception handling would grow exponentially and become less readable.

Other authors have also shown that it is possible to use sets of pre-defined recovery patterns to heal a system. For instance, the authors of [Chang2009] defined several healing patterns in order to evaluate the effectiveness of their *healing connectors*¹ mechanism. These healing patterns were implemented on the healing connectors used for testing purposes with applications such as Apache ActiveMQ [Apache2009b], Apache Service Mix [Apache2009c], JBoss [JBoss2009], among others. The problems that these healing patterns try to eliminate were carefully selected from the lists of bugs available on-line for each one of these projects. In a way, these patterns can be considered very similar to our recovery actions. Furthermore, the manner they were produced, by collecting bug-reports and repair information available on-line, can also be used for the development of system level recovery actions. Table 5.7 was adapted from [Chang2009] and shows 5 healing strategies of a total of 31 that were proven valid on the cited work.

Table 5.7 – Healing strategies.

Application	COTS component where failure originates	Failure description	Healing strategy
Apache Geronimo	Sun JRE 1.6 (ClassLoader component)	At startup, a faulty implementation of <code>classloader.loadClass</code> raises an exception when used to load an array with name specified with array syntax	Substitute the invocation of <code>classLoader.loadClass()</code> with <code>Class.forName()</code>
JBoss	Sun JRE 1.6 (ClassLoader component)	At startup, a faulty implementation of <code>classloader.loadClass</code> raises an exception when used to load an array with name specified with array syntax	Substitute the invocation of <code>classLoader.loadClass()</code> with <code>Class.forName()</code>

¹ Healing connectors were already discussed on Section 4.3.

Application	COTS component where failure originates	Failure description	Healing strategy
Developer application reproduced from bug report GERONIMO-1669	Apache Geronimo (JavaMail component)	Disconnected smtp transport raises an exception when sending a mail Message	Call the transport connect() operation before re-invoking the send the message
Apache ActiveMQ	Sun JRE (JavaNet component)	Starting ActiveMQ raises exceptions when hostname contains underscore characters	Replace the hostname string parameter by its IP address, and re-invoke the original operation
Apache ServiceMix	Sun JRE (JavaNet component)	Starting ServiceMix raises exceptions when hostname contains underscore characters	Replace the hostname string parameter by its IP address, and re-invoke the original operation
Magnolia CMS	Xalan XSLT	Magnolia cannot run and raises exceptions when initializing its content repositories	Dynamically load the jar files, delete the corrupted repository directories, and re-invoke the original operation

5.4. The Perfect Exception Handling Model

Garcia et al. [Garcia2001] have given us the criteria to evaluate the quality of an exception handling model in terms of reliability. At the same time, they were also able to provide us with a set of quality metrics to help guiding the development of future exception handling models. In

Table 5.8 we confront the attributes of our model with the desirable attributes of Garcias’s “perfect” model for exception handling.

Our AEH model is:

- *Modular* – In our system, recovery code is included as a *plug-in* component. Recovery blocks can be added, removed or modified on the deployed system without requiring changes to the remaining facilities;
- *Reusable* – The recovery code is intended to be portable across applications and platforms;

Table 5.8 – Exception model features list.

	AEH Model
Exceptions represented as objects	Yes
Exception list	Yes
Internal exceptions should be differentiated from external exceptions	No ¹
Avoid the definition of code blocks for the sole purpose of attaching an handler	Yes
Allow an hybrid binding (dynamic + static) of exception handlers	Yes
Allow for both explicit (one-level) and automatic propagation of exceptions	Yes
Perform automatic clean-up actions	Yes (transactions)
Implement the termination model	Yes
Allow for static and dynamic reliability checks	Yes
Fully support concurrent and distributed models	No

- *Maintainable* – Modifications on recovery policies can be done in a centralized manner;
- *Reliable* – As shown on this chapter, system reliability is prone to increase when compared with existent software using the traditional approach;
- *Simple* – The programmer can concentrate on writing the business logic code. The binding between business code and error recovery code can be completely transparent or customized with the aid of specialized visual tools;
- *Uniform* – Recovery actions can be made global to a program, platform, or system. The programming model is the same independently of the abstraction level. Also, the expected behavior on the presence of an error is common to all components;
- *Easily testable* – Recovery code can be tested separately from business code. Tests can concentrate on business code and its integration with the general recovery policies;

¹ On object-oriented systems we consider that this characteristic can be mimicked by the use of inheritance.

- *Traceable* – The execution of the recovery actions is the responsibility of the execution platform and it is controllable by the usage of configuration files. Monitoring is also privileged under this conditions;
- *Increases code readability* – Business logic code is separated from error handling code. There is less code to read and less control flow paths to understand;
- *Simplifies code writing* – In most cases, the programmer can concentrate on writing business code alone.

5.5. Summary

In this chapter we described and tested an implementation of the Automatic Exception Handling Model. Our framework allows the programmer to use the programming model proposed on Chapter 4 and provides the entire model's functionality.

We discussed the architecture of the development/execution framework and detailed the implementation of two core components: the STM library and the AEH Class Loader.

The STM library contains the basic functions and types associated with transactional model. The AEH Class Loader modifies the classes being loaded into the runtime environment by inserting the new transactional and recovery code.

We discussed the binding of exceptional occurrences with their respective system recovery actions. We have shown that configuration files can be quite large and complex to the human reader. Thus, we proposed and exemplified two mechanisms (tools) that simplify and aid on the creation of configuration files while coding.

On the second half of the chapter we validated the exception handling model and evaluated its implementation on four distinct vectors: a) amount of source code written by the programmer; b) effect on the application's resilience; c) performance penalty imposed by the exception runtime; d) viability of providing general recovery actions. Our results are very promising. We obtained a substantial decrease on the amount of exception handling that has to be written (less 30%), program's reliability improved, the performance penalty is negligible, and we were able to propose recovery actions for more than 60% of the exception types we analyzed.

We are convinced that an exception handling model evaluation can only be done on “the field”. The synthetic tests that we performed on this chapter are not sufficient for justifying the adoption of the model per se. The major problem is the lack of critical mass, this kind of proposal has to be used and evaluated by thousand of developers before definitive conclusions can be taken. Only when many companies, software designers, and programmers start using the new model on their development process, its true qualities and shortcomings will surface. Nonetheless, our validation efforts show that the model is feasible, can be implemented and incorporated into *production-state* development frameworks, can contribute to increase the quality of programs’ code, increase software reliability, and, at the same time, lower development times.

Chapter

6

Conclusion

This is the final chapter of the dissertation and it provides an overview of the work done, the problems that have been addressed and the contributions to the current state of the technology. A perspective on possible future work is also given.

6.1. Overview

Exception handling mechanisms are the *de facto* mechanism for error handling on modern object-oriented programming languages. Exceptions provide an elegant and civilized way of dealing with abnormal events. Nevertheless, the mechanism has flaws and some pose as threats for the reliability of programs and systems.

In this thesis we identified the major design shortcomings associated with existent exception handling models. We have shown how those weaknesses affect the way programmers are writing exception handling code today, and how, ultimately, they affect the quality of error recovery code and the resilience of programs to errors.

The main objective of this work was to propose a new exception handling model and demonstrate how it successfully mitigates some of the problems with current exception handling models. Our model deals with exceptions automatically. Thus, exceptions become much more a platform issue rather than the programmer's responsibility. This ultimately contributes to increase applications resilience to errors because, as we have shown, programmers are neglecting exception handling code and an automated approach currently offers better guarantees of recovery than the code that programmers typically write. Writing error recovery code is not a major concern for nowadays developers and consequently exception handling code quality is very low.

Our ambition for the automatic exception handling model is that it will some day represent to programs reliability what garbage collectors symbolize to memory management. Our model automates not only recovery code and its execution, but also the clean-up of the effects of unsuccessful executions. To achieve such a goal, we resort to a transactional mechanism that controls the execution of the protected regions on the code (try blocks) and of recovery actions. The proposed model mingles the concepts of traditional exception handling, software transactional memory and recovery blocks in order to achieve its objectives.

We described the architecture of the automatic exception handling model on Chapter 4 and discussed a possible implementation on Chapter 5. We also conducted several tests in order to evaluate both the proposed model and its implementation. Our experiences involved the modification of several applications in order to make them use the new exception handling approach. Results are very promising. There was a substantial decrease on the amount of exception handling that had to be written on the selected test programs;

the programs reliability was improved on the new versions; and the performance penalty was negligible. Furthermore, we showed that it is possible to define automatic recovery actions for a large number of exception types on the Java platform.

In general, we are confident that the proposed model represents a step forward in terms of software reliability for object-oriented programming languages. We feel that the automatic exception handling model fulfilled its objectives, showing that it is possible to improve software resilience while effectively decreasing the amount of exception handling code that programmers have to write.

6.2. Contributions

The major contributions of this dissertation can be summed up as:

- Providing a clear assessment of how programmers are using the existent exception handling mechanisms on their programs, exposing the problems behind the lack of quality of exception handling code, and identifying common bad practices, thus providing a clear base to guide the development of future exception handling models;
- Providing an exception handling model that is able to comply with the developers natural tendency to write business logic code without the “distraction” of considering exceptional situations. A model that it is able to increase the resilience of programs to abnormal situations even when developers pay no attention to error recovery, and which has the potential to eliminate many known exception programming bad practices. Thus, overall a model that simplifies the task of producing reliable code, with automatic recovery and clean-up, in a transparent way without resorting to complex language artifacts;
- To demonstrate that it is possible to integrate the new model into a production platform, such as the Java platform, in a simple way, with minimal lexical changes to an object-oriented programming language and a negligible performance overhead.

Recently, we witnessed, from the programming languages and models community, a growing effort to understand why programmers neglect exception handling [Shah2008a] and why changes are occurring on the way exception handling constructs are used. We

know that exception handling practice has shifted from an error recovery approach to a debugging approach (*exception detection->log->program termination*). Exceptions are no longer being used to provide proper error recovery. Many platform designers, such as .NET designers, support this move. They believe that latent errors should be eliminated through exhaustive testing. We, on the other hand, consider that programming techniques are constantly evolving, and, from our point of view, the correct path is creating frameworks that can offer a high degree of resilience to errors even when developers give no special attention to the subject. And if the programmer prefers to focus on writing program business logic code without immediately having to consider the abnormal cases that might occur, he or she should not be forced to do the opposite. Nonetheless, such power must come without jeopardizing the resilience of programs to errors.

6.3. Future Work

Future work, from our point of view, poses as three distinct challenges:

- Develop a production ready framework with all the necessary tools to write, compile, test, deploy, execute and maintain software using the automatic exception handling model. This includes creating a complete set of recovery actions for system (and system libraries) exception types to deploy with the platform;
- Conduct experiments that can help understand how developers relate to the new model in respect to the traditional approaches;
- Explore the model's applicability to the area of concurrent programming upgrading the model to allow the execution of cooperative and concurrent automatic error recovery actions.

Exception handling plays a vital role in the overall reliability of software. It is an often neglected design area of programming languages that has a crucial impact on the overall quality and robustness of applications. Due to the increasing demand for easier ways of dealing with abnormal events, the ubiquity of concurrent programming and the need to provide end-users with more reliable software, the research activity in this area has been steadily increasing. Furthermore, it is also interesting to see modern programming languages and platforms incorporating more advanced and sophisticated ways of dealing

with errors. In the future, we hope to continue to contribute to the state-of-the-art on exception handling and help support the growth of this important research area.

List of Publications

The following publications are the result of the investigation undertaken during this thesis.

Publications on journals

- B. Cabral, P. Marques, *"A Model For Automatic Exception Handling"* (currently submitted to IEEE Transactions on Software Engineering)
- B. Cabral, P. Sacramento, P. Marques, *"The Hidden Truth Behind .NET Exception Handling Today"*, in IET Software Journal, Vol. 1(6), pp. 223-250, IET, December 2007.
- B. Cabral, P. Marques, L. Silva, *"IL Code Instrumentation with RAIL"*, in .NET Developers Journal, Vol. 2(1), pp. 34-35, SYS-CON Media Publishers, January 2004.

Publications on conference proceedings

- B. Cabral, P. Marques, *"Implementing Retry – featuring AOP"*, in Proc. of the Latin-American Symposium on Dependable Computing (LADC'09), IEEE Computer Society Press, João Pessoa, Paraíba, Brazil, September 2009.
- B. Cabral, P. Marques, *"A Case For Automatic Exception Handling"*, in Proc. of the 23rd IEEE/ACM International Conference on Automated Software Engineering, IEEE Computer Society Press, L'Aquila, Italy, September 2008.
- B. Cabral, P. Marques, *"Exception Handling: A field study in Java and .NET"*, in Proc. of the European Conference in Object-Oriented Programming 2007 (ECOOP'07), Berlin, Germany, July 2007.
- B. Cabral, P. Marques, *"Making Exception Handling Work"* (extended abstract), in Proc. of the USENIX Second Workshop on Hot Topics in System Dependability (HotDep'06), USENIX, Seattle, USA, November 2006.
- P. Sacramento, B. Cabral, P. Marques, *"Unchecked Exceptions: Can the Programmer be Trusted to Document Exceptions?"*, in Proc. of the International Conference on

Innovative Views of .NET Technologies (IVNET'06), Springer-Verlag, October 2006. (*Best Paper Award*)

- B. Cabral, P. Marques, L. Silva, "*RAIL: Code Instrumentation for .NET*", in Proc. of the 2005 ACM Symposium On Applied Computing (SAC'05), ACM Press, Santa Fé, New Mexico, USA, March 2005.
- B. Cabral, P. Marques, L. Silva, "*RAIL: Code Instrumentation for .NET*" (extended abstract), in Proc. of the ACM OOPSLA'04 Conference Companion, ACM Press, Vancouver, Canada, October 2004.

Bibliography

- [Allan2005] E. Allan, D. Chase, V. Luchangco, J. Maessen, S. Ryu, G. Steele, and S. Tobin-Hochstadt, *"The Fortress language specification version 0.785. Technical report"*, Sun Microsystems, 2005.
- [Alonso1994] G. Alonso, M. Kamath, D. Agrawal, A. E. Abbadi, R. Gunthor, C. Andmohan, *"Failure handling in large-scale workflow management systems"*, Technical Report RJ9913, IBM Almaden Research Center, San Jose, CA, November 1994.
- [Anderson1975] T. Anderson, *"Provably Safe Programs"*, Technical Report 70, Computing Laboratory, University of Newcastle upon Tyne, 1975.
- [Apache2009a] The Apache Software Foundation, *"Tomcat 6"*, 2009, available at: <http://tomcat.apache.org/> .
- [Apache2009b] The Apache Software Foundation, *"ActiveMQ"*, 2009, available at: <http://activemq.apache.org/> .
- [Apache2009c] The Apache Software Foundation, *"ServiceMix"*, 2009, available at: <http://servicemix.apache.org/home.html> .
- [Apache2009d] The Apache Software Foundation, *"JMeter"*, 2009, available at: <http://jakarta.apache.org/jmeter/> .
- [Arnold2000] K. Arnold, J. Gosling, and D. Holmes, *"The Java Programming Language"*, 3rd Edition, Addison-Wesley Longman Publishing Co., Inc., 2000.
- [Avizienis1977] A. Avizienis, and L. Chen, *"On the implementation of N-version programming for software fault tolerance during execution"*, in Proceedings of IEEE COMPSAC 77, November 1977.
- [Ball2001] T. Ball, and S. K. Rajamani, *"Automatically validating temporal safety properties of interfaces"*, in SPIN 2001, Workshop on Model Checking of Software, LCNS 2057, Springer-Verlag, New York, 2001.

- [Balter1994] R. Balter, S. Lacourte, and M. Riveill, "*The Guide Language*", in *The Computer Journal*, Vol. 37(6), British Computer Society, 1994.
- [Best1996] E. Best, "*Semantics of Sequential and Parallel Programs*", Prentice Hall, Upper Saddle River, NJ, USA, 1996.
- [Brown1998] W. J. Brown, R. C. Malveau, H. W. McCormick, T. J. Mowbray, "*Antipatterns: Refactoring Software, Architectures, and Projects in Crisis*", 1st Edition, Wiley, March 1998.
- [Bruneton2002] E. Bruneton, R. Lenglet, and T. Coupaye, "*ASM: a code manipulation tool to implement adaptable systems*", in *ACM SIGOPS Adaptable and Extensible Component Systems*, ACM Press, Grenoble, France, November 2002.
- [Burh1992] P. A. Buhr, H. I. Macdonald, and C. R. Zarnke, "*Synchronous and asynchronous handling of abnormal events in the System*", *Software: Practice and Experience*, Vol. 22(9), September 1992.
- [Burh2000] P. A. Buhr, and W. Y. Mok, "*Advanced Exception Handling Mechanisms*", in *IEEE Transactions on Software Engineering*, Vol. 26(9), IEEE, September 2000.
- [Cabral2005] B. Cabral, P. Marques, and L. Silva, "*RAIL: Code Instrumentation for .NET*", in *Proceedings of the 2005 ACM Symposium On Applied Computing (SAC'05)*, ACM Press, Santa Fé, New Mexico, USA, March 2005.
- [Cabral2006] B. Cabral, P. Marques, "*Making Exception Handling Work*" (extended abstract), in *Proceedings of the USENIX Second Workshop on Hot Topics in System Dependability (HotDep'06)*, USENIX, Seattle, USA, November 2006.
- [Cabral2007] B. Cabral and P. Marques, "*Exception Handling: A Field Study in Java and .NET*", in *Proceedings of the 21st European Conference on Object-Oriented Programming (ECOOP '07)*, LNCS 4609, Springer-Verlag, 2007.

- [Cabral2007b] B. Cabral, P. Sacramento, P. Marques, "*The Hidden Truth Behind .NET Exception Handling Today*", in IET Software Journal, Vol. 1(6), pp. 223-250, IET, December 2007.
- [Cabral2008] B. Cabral, P. Marques, "*A Case For Automatic Exception Handling*", in Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering, IEEE Computer Society Press, L'Aquila, Italy, September 2008.
- [Cabral2009] B. Cabral, P. Marques, "*Implementing Retry - featuring AOP*", in Proceedings of the Latin-American Symposium on Dependable Computing (LADC'09), , IEEE Computer Society Press, João Pessoa, Paraíba, Brazil, September 2009.
- [Campbell1983] R. H. Campbell, T. Anderson, B. Randell, "*Practical Fault Tolerance Software for Asynchronous Systems*", in IPAC Safecom'83, Cambridge, UK, 1983.
- [Campbell1986] R. H. Campbell, B. Randell, "*Error Recovery in Asynchronous Systems*", in IEEE Transactions on Software Engineering, Vol. SE-12(8), 1986.
- [Carlstrom2006] B. D. Carlstrom, A. McDonald, H. Chafi, J. Chung, C. C. Minh, C. Kozyrakis, and K. Olukotun, "*The Atomos transactional programming language*", SIGPLAN Notices Vol. 41(6), Jun. 2006.
- [Caseu1987] Y. Caseau, "*Etude et realisation d'un langage objet: LORE*", PhD dissertation, Paris XI University, Orsay, November 1987.
- [Charles2005] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, "*X10: an object-oriented approach to non-uniform cluster computing*", in ACM Conference on Object Oriented Programming Systems Languages and Applications, 2005.
- [Chang2009] H. Chang, L. Mariani, and M. Pezze, "*In-field healing of integration problems with COTS components*", in Proceedings of the 2009 IEEE 31st International Conference on Software Engineering, IEEE CS Press, Washington, DC, 2009.

- [Cheng2005] Y. C. Cheng, C. Chen, and J. Jwo, "*Exception Handling: An Architecture Model and Utility Support*", in Proceedings of the 12th Asia-Pacific Software Engineering Conference (APSEC'05), Taiwan, China, December, 2005.
- [Chiba2000] S. Chiba, "*Load-Time Structural Reflection in Java*", in Proceedings of the European Conference on Object-Oriented Programming (ECOOP'00), Springer-Verlag, LNCS 1850, Sophia Antipolis and Cannes, France, June 2000.
- [CodeProj2008] "*The Code Project - Free Source Code and Tutorials*", available at: <http://www.codeproject.com/>.
- [CpSphere2008] "*CpSphere Email Component for .NET*", available at: <http://www.codeproject.com/dotnet/cpSphereEmailComponent.asp>
- [Cristian1979] F. Cristian, "*A recovery mechanism for modular software*", in Proceedings of the 4th International Conference on Software Engineering, Piscataway, IEEE Press, New Jersey, 1979.
- [Cristian1980] F. Cristian, "*Exception Handling and Software Fault Tolerance*", In Proceedings of FTCS-25, 3, IEEE, 1996 (reprinted from FTCS-IO 1980, 97-103).
- [Chistian1995] F. Cristian, "*Exception handling and tolerance of software faults*", In Software Fault Tolerance, Wiley, 1995.
- [Cui1992] Q. Cui, and J. Gannon, "*Data-Oriented Exception Handling*", in IEEE Transactions on Software Engineering, Vol. 18(5), May 1992.
- [Dahl1972] O-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, "*Structured Programming*", Academic Press, New York, N.Y., 1972.
- [Dahm1999] M. Dahm, "*Byte Code Engineering*", in Java-Informationen-Tage, Springer-Verlag, Dusseldorf, Germany, September 1999.
- [Deline2001] R. Deline, and M. Fahndrich, "*Enforcing high-level protocols in low-level software*", in Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation, June 2001.

- [Dijkstra1968] E. W. Dijkstra, "*The structure of the 'THE'-multiprogramming system*", in *Communications of the ACM*, Vol. 11(5), ACM Press, May 1968.
- [Doshy2003] G. Doshy, "*Best Practices for Exception Handling*", in ONJava.com, O'Reilly, 2003, accessed 2008/04/01, available at: <http://www.onjava.com/pub/a/onjava/2003/11/19/exceptions.html>.
- [Eastlake1968] D. Eastlake, R. Greenblatt, J. Holloway, T. Knight, and S. Nelson, "*ITS 1.5 Reference Manual*", in AI Memo 161, MIT Artificial Intelligence Laboratory, Cambridge, Massachusetts, June 1968. Revised as Memo 161A, July 1969.
- [Eastlake1972] D. Eastlake, "*ITS Status Report*", in AI Memo 238, MIT Artificial Intelligence Laboratory, Cambridge, Massachusetts, April 1972.
- [Elrad2001] T. Elrad, R. E. Filman, and A. Bader, "*Aspect-Oriented Programming*", in *Communications of the ACM*, Vol. 44(10), ACM Press, New York, New York, USA, October 2001.
- [Fetzer2004] C. Fetzer, P. Felber, and K. Hogstedt, "*Automatic detection and masking of nonatomic exception handling*", in *IEEE Transactions on Software Engineering*, IEEE Press, 2004.
- [Fetzer2007] C. Fetzer and P. Felber, "Improving Program Correctness with Atomic Exception Handling", in *Journal of Universal Computer Science*, Vol. 13(8), 2007.
- [Filho2005] F. Filho, C. Rubira, and A. Garcia, "*A Quantitative Study on the Aspectization of Exception Handling*", in *Workshop on Exception Handling in Object-Oriented Systems (held in ECOOP 2005)*, Glasgow, Scotland, July 2005.
- [Filho2006] F. C. Filho, N. Cacho, E. Figueiredo, R. Maranhão, A. Garcia, and C. M. F. Rubira, "*Exceptions and aspects: the devil is in the details*", in *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ACM Press, New York, NY, USA, 2006.

- [Filho2007] F. C. Filho, A. Garcia, and C. M. F. Rubira, "*Error handling as an aspect*", in Proceedings of the 2nd workshop on Best Practices in Applying Aspect-Oriented Software Development, ACM Press, New York, NY, USA, 2007.
- [Gamma1995] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, "*Design patterns: elements of reusable object-oriented software*", Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.
- [Garcia2001] A. Garcia, C. Rubira, A. Romanovsky, and J. Xu, "*A Comparative Study of Exception Handling Mechanisms for Building Dependable Object-Oriented Software*", in The Journal of Systems and Software, Vol. 59(2), 2001.
- [GarciaMol1987] H. Gracia-Molina, and K. Salem, "*Sagas*", in Proceedings of the ACM Conference on Management of Data, ACM Press, New York, 1987.
- [Garthwaite1998] A. Garthwaite and S. Nettles, "*Transactions for Java*", in Proceedings of the International Conference in Computer Languages, IEEE Press, Chicago, Illinois, USA, 1998.
- [Gehani1992] N. H. Gehani, "*Exceptional C or C with Exceptions*", Software: Practice and Experience, Vol. 22(10), October 1992.
- [Goldberg1989] A. Goldberg, and D. Robson, "*Smalltalk-80: The Language*", Addison-Wesley, 1989.
- [Golomb1965] S. W. Golomb, and L. D. Baumert, "*Backtrack programming*", Journal of ACM, Vol. 12(4), October 1965.
- [Goodenough1975] J. B. Goodenough, "*Exception handling: issues and a proposed notation*", In Communications of the ACM, Vol. 18(12), ACM Press, December 1975.
- [Gosling2005] J. Gosling, B. Joy, G. Steele, and G. Bracha, "*The Java(TM) Language Specification*", 3rd Edition, Prentice Hall, June 2005.
- [Gray1993] J. N. Gray, A. Reuter, "*Transaction Processing: Concepts and Techniques*", Morgan Kaufmann, San Mateo, California, 1993.

- [Gunnerson2000] E. Gunnerson, "*C# and exception specifications*", Microsoft, 2000, available at:
<http://discuss.develop.com/archives/wa.exe?A2=ind0011A&L=DOTNET&P=R32820> .
- [Hamming1950] R. W. Hamming, "*Error Detecting and Error Correcting Codes*", In The Bell System Technical Journal, American Telephone and Telegraph Company, New York, April 1950.
- [Haines1994] N. Haines, D. Kindred, J. G. Morrisett, A. M. Nettles, J. M. Wing, "*Composing First-Class Transactions*", in ACM Transactions on Programming Languages and Systems, Vol. 16(6), 1994.
- [Halstead1985] R. Halstead, "*Multilisp: a language for concurrent symbolic computation*", in ACM Transactions on Programming Languages and Systems, Vol. 7(4), 1985.
- [Hindman2006] B. Hindman and D. Grossman, "*Atomicity via source-to-source translation*", in The 2006 workshop on Memory system performance and correctness, ACM Press, 2006.
- [Hoare1973] C. A. R. Hoare, "*Parallel programming: an axiomatic approach*", STAN-CS-73-394, AD769674, Department of Computer Science, Stanford University, October 1973.
- [Horning1974] J. J. Horning, H. C. Lauer, P. M. Melliar-Smith and B. Randell, "*A Program Structure for Error Detection and Recovery*", in Proceedings of the Conference on Operating Systems, LCNS, Vol. 16, Springer-Verlag, London, 1974.
- [IBM1968] International Business Machines (IBM), "*IBM System/360 PL/I Reference Manual*", SRL Form C28-8201-1, 1968.
- [IBM1981] International Business Machines (IBM), "*OS and DOS PL/I Reference Manual*", 1st Edition, September 1981.
- [IBM2008] International Business Machines (IBM), Jikes Compiler Homepage, available at: <http://jikes.sourceforge.net/>.

- [Issarny1993] V. Issarny, "An Exception Handling Mechanism for Parallel Object-Oriented Programming: Towards Reusable, Robust Distributed Software", in *Journal of Object-Oriented Programming*, Vol. 6(6), 1993.
- [ISO23270:2006] ISO/IEC, "Information Technology – Programming Languages - C#", 2nd Edition, ISO/IEC, Switzerland, 2006.
- [ISO23271:2006] ISO/IEC, "Information Technology – Common Language Infrastructure (CLI) Partition I to VI", 2nd Edition, ISO/IEC, Switzerland, 2006.
- [ISO8652:1995] Intermetrics (Ed.), "Information Technology - Programming Languages - Ada", ISO/IEC 8652:1995(E), 1995.
- [Jalote1994] P. Jalote, "Fault Tolerance in Distributed Systems", Prentice-Hall, Inc, Upper Saddle River, NJ, USA, 1994.
- [Javacc2008] Javacc - Java Compiler Compiler, available at: <https://javacc.dev.java.net/>.
- [JBoss2009] JBoss Community, "JBoss", 2009, available at: <http://www.jboss.org/about.html>.
- [Jimenez2000] R. Jimenez-Peris, M. Patino-Martinez, S. Arevalo, "TransLib: An Ada 95 Object Oriented Framework for Building Transactional Applications", in *Computer Systems: Science & Engineering Journal*, Vol. 15(1), 2000.
- [JSR166] JSR166: Concurrency utilities, Sun Microsystems, available at: <http://java.sun.com/j2se/1.5.0/docs/guide/concurrency>.
- [JWAM2008] JWAM web site, 2008, available at: [http://www.c1-wps.de/nc/loesungen/jwam/?sword_list\[0\]=jwam](http://www.c1-wps.de/nc/loesungen/jwam/?sword_list[0]=jwam).
- [Kienzle2001a] J. Kienzle, A. Romanovsky, "Combining Tasking and Transactions: Open Multithreaded Transactions", *Ada Letters*, Vol. 21(1), 2001.
- [Kienzle2001b] J. Kienzle, A. Strohmeier, and A. Romanovsky, "Open Multithreaded Transactions: Keeping Threads and Exceptions under Control", in *Proceedings of the Sixth International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS'01)*, IEEE Computer Society, Los Alamitos, CA, USA, 2001.

- [Kiczales1997] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin, "*Aspect-Oriented Programming*", in Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP '97), Springer-Verlag, Finland, 1997.
- [Kimmel2001] P. Kimmel, "*Delphi 6 Developer's Guide*", McGraw-Hill School Education Group, 2001.
- [Knowgate2006] Knowgate, "*Hipergate: Free CRM Groupware and Intranet Software*", 2009, available at: <http://www.hipergate.org/> .
- [Knudsen1984] J. L. Knudsen, "*Exception handling - a static approach*", in Software: Practice and Experience, Vol. 14(5), May 1984.
- [Knudsen1987] J. L. Knudsen, "*Better exception handling in block structured systems*", in IEEE Software, Vol. 4(3), May 1987.
- [Koenig1990] A. Koenig, and B. Stroustrup, "*Exception handling in C++*", in the Journal of Object-Oriented Programming, Vol. 3(2), July/August 1990.
- [Koenig1993] A. Koenig, B. Stroustrup, "*Exception handling for C++*", in The evolution of C++: language design in the marketplace of ideas, MIT Press, 1993.
- [Korth1990] H. F. Korth, E. Levy, and A. Silberschatz, "*A formal approach to recovery by compensating transactions*", Technical Report, UMI Order Number: CS-TR-90-14., University of Texas at Austin.
- [KurkiSuonio1997] R. Kurki-Suonio, T. Mikkonen, "*Liberating object-oriented modeling from programming-level abstractions*", in Proceedings of the Workshops on Object-Oriented Technology, LNCS 1357, Springer-Verlag, London, 1997.
- [Lang1998] J. Lang, D. Stewart, "*A Study of the Applicability of Existing Exception Handling Techniques to Component-Based Real-Time Software Technology*", ACM Transactions on Programming Languages Systems, Vol. 20(2), March 1998.

- [Lanvin2009] D. Lanvin, R. I. Castanedo, A. Fuente, A. Álvarez, "Extending object-oriented languages with backward error recovery integrated support", in *Computer Languages, Systems & Structures*, Elsevier Ltd., May 2009.
- [Laprie1987] J. C. Laprie, J. Arlat, C. Beounes, K. Kanoun, and C. Hourtolle, "Hardware and software fault-tolerance: definition and analysis of architectural solutions", in *Proceedings of the 17th International Symposium on Fault-Tolerant Computing*, IEEE, 1987.
- [Laprie1990] J. C. Laprie, J. Arlat, C. Beounes, K. Kanoun, "Definition and Analysis of Hardware- and Software-Fault-Tolerance Architectures", *IEEE Computer*, Vol. 23(7), 1990.
- [Laprie1995a] J. C. Laprie, J. Arlat, C. Béounes, K. Kanoun, "Architectural Issues in Software Fault Tolerance", *Software Fault Tolerance*, John Wiley & Sons, 1995.
- [Laprie1995b] J. C. Laprie, "Dependable Computing: Concepts, Limits, Challenges", in *Proceedings of the 25th IEEE International Symposium on Fault-Tolerant Computing - Special Issue*, Pasadena, California, USA, IEEE, 1995.
- [Lemos2001] R. de Lemos, and A. B. Romanovsky, "Exception Handling in the Software Lifecycle", in *International Journal of Computer Systems Science and Engineering*, Vol. 16(2), 2001.
- [Levin1977] R. Levin, "Program structures for exceptional condition handling", Ph.D. dissertation, Dep. Computer Science, Carnegie-Mellon University, Pittsburgh, PA, June 1977.
- [Limewire2009] Lime Wire LLC, "Limewire", 2009, available at: <http://www.limewire.com>.
- [Lippert2000] M. Lippert, C. V. Lopes, "A Study on Exception Detection and Handling Using Aspect-Oriented Programming", in *Proceedings of the 22nd International Conference on Software Engineering (ICSE 2000)*, ACM Press, New York, NY, USA, 2000.

- [Liskov1972a] B. H. Liskov, "A design methodology for reliable software systems", in Proceedings of the AFIPS Fall Joint Computer Conference, Vol. 41(1), AFIPS Press, Montvale, N. J., 1972.
- [Liskov1972b] B. H. Liskov, "The design of the VENUS operating system", in Communications of the ACM, Vol. 15(3), ACM Press, March 1972.
- [Liskov1974] B. H. Liskov, and S. Zilles, "Programming with abstract data types", in SIGPLAN Notices (ACM Newsletter), Vol. 9(4), April 1974.
- [Liskov1979] B. H. Liskov, and A. Snyder, "Exception Handling in CLU", in IEEE Transaction on Software Engineering, Vol. SE-5(6), 1979.
- [Liskov1988] B. H. Liskov, "Distributed Programming in Argus", in Communications of the ACM, Vol. 31(3), 1988.
- [Lopes1998] C. Lopes, G. Kiczales, "Recent Developments in AspectJ", in European Conference on Object-Oriented Programming (ECOOP '98) Workshop Reader, Springer-Verlag, 1998.
- [Madsen1993] O. Madsen, B. Møller-Pedersen, and K. Nygaard, "Object-oriented Programming in the BETA Programming Language", Addison-Wesley, 1993.
- [Malayeri2006] D. Malayeri, and J. Aldrich, "Practical exception specifications", in Advanced Topics in Exception Handling Techniques, LCNS 4119, Springer-Verlag, New York, 2006.
- [Martin2006] M. Martin, C. Blundell and E. Lewis, "Subtleties of Transactional Memory Atomicity Semantics", in IEEE Computer Architecture Letters, Vol. 5(2), IEEE Press, 2006.
- [McCarthy1965] J. McCarthy, P. W. Abrams, D. J. Edwards, T. P. Hart, and M. Levin, "LISP 1.5 Programmers Manual", MIT Press, 1965.
- [McConnel2004] S. McConnel, "Code Complete", 2nd Edition, Microsoft Press, Redmond, WA, USA, 2004.
- [McCune2006] T. McCune, "Exception-Handling Antipatterns", in Java.net, Sun Microsystems, Inc., O'Reilly, and Collabnet, 2006, available at:

<http://today.java.net/pub/a/today/2006/04/06/exception-handling-antipatterns.html> .

- [Meyer1988] B. Meyer, *"Object-Oriented Software Construction"*, Prentice-Hall, 1988.
- [Microsoft2008] *Microsoft Visual Studio 2008*, Microsoft Corporation, available at: <http://msdn.microsoft.com/en-us/vstudio/default.aspx>.
- [Miller1997] R. Miller and A. Tripathi, *"Issues with exception handling in object-oriented systems"*, in Proceedings of European Conference on Object-Oriented Programming (ECCOP '97), LCNS 1241, Springer-Verlag, 1997.
- [Miller2002] R. Miller, A. Tripathi, *"The guardian model for exception handling in distributed systems"*, in Proceedings of the 21st IEEE Symposium on Reliable Distributed Systems (SRDS 2002), IEEE, Piscataway NJ, USA, 2002.
- [Mitchel1979] J. G. Mitchell, W. Maybury, and R. Sweet, *"Mesa language manual"*, Technical Report CSL-79-3, Xerox Palo Alto Research Center, April 1979.
- [Mok1997] W. Y. R. Mok, *"Concurrent abnormal event handling mechanisms"*, M.S. thesis, University of Waterloo, Waterloo, Ontario, Canada, N2L 3G1, September 1997, available at: <ftp://plg.uwaterloo.ca/pub/uSystem/-MokThesis.ps.gz>.
- [Mono2008] *"Mono"*, available at: <http://www.go-mono.com>.
- [Moon1974] D. A. Moon, *"MacLISP Reference Manual"*, MIT Project MAC, Cambridge, Massachusetts, April 1974.
- [Moss2006] J. Moss and A. Hosking, *"Nested Transactional Memory: Model and Architecture Sketches"*, in Science of Computer Programming, Vol. 63(2), Elsevier Science, December 2006.
- [Motet1996] G. Motet, A. Mapinard, and J. C. Geoffroy, *"Design of Dependable Ada Software"*, Prentice Hall, 1996.

- [Muller2002] A. Müller and G. Simmons, "Exception Handling: Common Problems and Best Practice With Java 1.4", in Proceedings of Net.ObjectDays 2002, Erfurt, Germany, October 2002.
- [NAnt2008] "NAnt - A .NET Build Tool", available at: <http://nant.sourceforge.net>.
- [NDoc2008] "NDoc Code Documentation Generator for .NET", available at: <http://ndoc.sourceforge.net>.
- [Nelson1991] G. Nelson (Ed.), "System Programming with Modula-3", in Prentice Hall Series in Innovative Technology, ISBN 0-13-590464-1, L.C. QA76.66.S87, 1991.
- [OMG1996] Object Management Group Object Transaction Service, Draft 4, OMG, OMG Document, 1996.
- [Oki1983] B. M. Oki, "Reliable object storage to support atomic actions", MSc thesis, MIT Department of EE and CS, May 1983.
- [Papurt1998] D. Papurt, "The Use of Exceptions", in the Journal of Object-Oriented Programming, 1998.
- [Parr2006] T. Parr, "ANTLR - Another Tool for Language Recognition", University of San Francisco, 2006, available at: <http://www.antlr.org/>.
- [Pezz2004] M. Pezz, M. Young, "Testing Object Oriented Software", in Proceedings of the 26th International Conference on Software Engineering (ICSE'04), 2004.
- [Plank1996] J. S. Plank, Y. Chen, K. Li, M. Beck, and G. Kingsley, "Memory Exclusion: Optimizing the Performance of Checkpointing Systems", in Software - Practice and Experience, Vol. 29(2), 1996.
- [Portland2007] Portland Exception Patterns Repository, accessed 2008/04/01, available at: <http://c2.com/cgi/wiki?ExceptionPatterns> .
- [Radin1981] G. Radin, "The early history and characteristics of PL/I", in History of Programming Languages, Academic Press, 1981.

- [Randell1975] B. Randell, "*System structure for software fault tolerance*", in Proceedings of the International Conference on Reliable Software, ACM Press, New York, NY, USA, 1975.
- [Randell1978] B. Randell, P. Lee, and P. C. Treleaven, "*Reliability Issues in Computing System Design*", in ACM Computing Surveys, Vol. 10(2), ACM Press, New York, NY, USA, 1978.
- [Randell1982] B. Randell, "*The Origins of Digital Computers: Selected Papers*", 3rd Edition, Springer-Verlag, Heidelberg, 1982.
- [Randell1995] B. Randell, A. Romanovsky, C. M. F. Rubira-Calsavara, R. J. Stroud, Z. Wu, and J. Xu, "*From Recovery Blocks to Concurrent Atomic Actions*", in Predictably Dependable Computing Systems, ESPRIT Basic Research Series, Springer-Verlag, Brussels, 1995.
- [Robillard2000] M. P. Robillard, and G. C. Murphy, "*Designing robust Java programs with exceptions*", in Proceedings of the 8th ACM SIGSOFT International Symposium on Foundations of Software Engineering, Vol. 25(6), ACM Press, November 2000.
- [Romanovsky1999] A. Romanovsky, "*On Structuring Cooperative and Competitive Concurrent Systems*", Computer Journal, Vol. 42(8), 1999.
- [Romanovsky2001] A. Romanovsky, and J. Kienzle, "*Action-Oriented Exception Handling in Cooperative and Competitive Concurrent Object-Oriented Systems*", In Advances in Exception Handling Techniques, LNCS-2022, Springer-Verlag, 2001.
- [Ross1967] D. T. Ross, "*The AED free storage package*", in Communications of the ACM, Vol. 10(8), August 1967.
- [Ryder2003] B. G. Ryder, M. L. Soffa, "*Influences on the design of exception handling: ACM SIGSOFT project on the impact of software engineering research on programming language design*", in SIGPLAN Notices, Vol. 38(6), ACM Press, 2003.
- [Sacramento2006] P. Sacramento, B. Cabral, and P. Marques, "*Unchecked Exceptions: Can the Programmer be Trusted to Document Exceptions?*", in Proceedings of

- the International Conference on Innovative Views of .NET Technologies (IVNET'06), Springer-Verlag, October 2006.
- [Scott1987] R. K. Scott, J. W. Gault, and D. F. McAllister, "*Fault-Tolerant Software Reliability Modeling*", in IEEE Transactions on Software Engineering, Vol. SE-13(5), May 1987.
- [Shah2008a] H. Shah, C. Görg, and M. J. Harrold, "*Why do developers neglect exception handling?*", in Proceedings of the 4th international Workshop on Exception Handling, WEH '08, ACM Press, New York, NY, 2008.
- [Shah2008b] H. Shah, C. Görg, and M. J. Harrold, "*Visualization of exception handling constructs to support program understanding*", in Proceedings of the ACM Symposium on Software Visualization, ACM Press, New York, NY, 2008.
- [SharpZipL2008] "*SharpZipLib, The Zip, GZip, BZip2 and Tar Implementation For .NET*", available at: <http://www.icsharpcode.net/OpenSource/SharpZipLib/Default.aspx>.
- [Shavit1995] N. Shavit, and D. Touitou, "*Software Transactional Memory*", in Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing, ACM Press, New York, New York, USA, 1995.
- [Shinnar2004] A. Shinnar, D. Tarditi, M. Plesko and B. Steensgaard, "*Integrating Support for Undo with Exception Handling*", Microsoft Research, December 2004.
- [Silva1997] A. Silva, J. Pereira, and J. Marques, "*Customizable Object Recovery Pattern*", Pattern Languages of Program Design 3, Addison-Wesley, 1997.
- [Sinha1999] S. Sinha, and M. J. Harrold, "*Criteria for testing exception-handling constructs in Java programs*", in Proceedings of the International Conference on Software Maintenance (ICSM'99), IEEE Computer Society, available at: <http://computer.org/proceedings/icsm/0016/0016toc.htm>, 265–276.

- [Sinha2000] S. Sinha, and M. Harrold, "*Analysis and Testing of Programs with Exception-Handling Constructs*", in IEEE Transactions on Software Engineering, Vol. 26(9), SEPTEMBER 2000.
- [Softec1972] SofTech, Inc, AED Programmer's Guide, SofTech, Inc., Waltham, Massachusetts, 1972.
- [SourceFrg2008] "*Sourceforge.net*", available at: <http://sf.net>.
- [Sun2004] Sun Microsystems, Inc., *Javadoc 1.5.0*, Sun Microsystems, Inc., available at: <http://java.sun.com/j2se/javadoc/>
- [Sun2006] Sun Microsystems, Inc., Enterprise JavaBeans Specification, v.3.0, Sun Microsystems, Inc. 2006.
- [Sun2008] Sun Microsystems, Inc., GlashFish Open Message Queue, available at: <http://java.sun.com/products/jms/>.
- [Sun2009] Sun Microsystems, Inc., "MySQL", 2009, available at: <http://www.mysql.com/>.
- [SSCLI2008] "*The Microsoft Shared Source CLI Implementation*", available at: <http://msdn.microsoft.com/net/sscli>.
- [Steele1993] G. L. Steele Jr., R. P. Gabriel, "*The Evolution of Lisp*", in ACM SIGPLAN Notices, Vol. 28, ACM Press, March 1993.
- [Stroustrup1994] B. Stroustrup, "*The Design and Evolution of C++*", Addison-Wesley, 1994.
- [Tennent1980] R. D. Tennent, "*Language design methods based on semantic principles*", in Acta Infomatica, Vol. 8(2), 1977. Reprinted in Tutorial: Programming Language Design, A. I. Wasserman (Ed.), Computer Society Press, 1980.
- [Tikhomirova1997] N. V. Tikhomirova, I. V. Shturtz, and A. Romanovsky, "*Object-oriented approach to state restoration by reversion in fault tolerant systems*", Computing Science, Newcastle, University of Newcastle upon Tyne, United Kingdom, 1997.

- [Utas2004] G. Utas, *“Robust Communications Software: Extreme Availability, Reliability and Scalability for Carrier-Grade Systems”*, Wiley, 2004.
- [Weimer2008] W. Weimer, and G. C. Necula, *“Exceptional situations and program reliability”*, in ACM Transactions on Programming Languages and Systems, Vol. 30(2), March 2008.
- [Wirfs-Brock2006] R. J. Wirfs-Brock, *“Towards Exception-Handling Best Practices and Patterns”*, in IEEE Software, Vol. 23 (5), IEEE Computer Society, September/October 2006.
- [Woodger1972] M. Woodger, *“On semantic levels in programming”*, in Proceedings of the IFIP World Computer Congress, North-Holland Pub. Co., Amsterdam, 1972.
- [Xu1995] J. Xu, B. Randell, A. Romanovsky, C. Rubira, R. Stroud, Z. Wu, *“Fault Tolerance in Concurrent Object-Oriented Software through Coordinated Error Recovery”*, in Proceedings of the 25th FTCS - International Symposium on Fault-Tolerant Computing, Pasadena, USA, 1995.
- [Xu1997] J. Xu, and B. Randell, *“Software Fault Tolerance: $t/(n-1)$ -Variant Programming”*, in IEEE Transactions on Reliability, Vol. 46(1), March 1997.
- [Yemini1985] S. Yemini, D. Berry, *“A Modular Verifiable Exception Handling Mechanism”*, in ACM Transactions on Programming Languages and Systems, Vol. 7(2), 1985.
- [Zhang2007a] L. Zhang, C. Krintz, and P. Nagpurkar, *“Supporting exception handling for futures in java”*, in Proceedings of the 5th International Symposium on Principles and Practice of Programming in Java (PPPJ '07), ACM Press, New York, NY, USA, 2007.
- [Zhang2007b] L. Zhang, C. Krintz, and P. Nagpurkar, *“Language and Virtual Machine Support for Efficient Fine-Grained Futures in Java”*, in the 16th International Conference on Parallel Architectures and Compilation Techniques (PACT), September 2007.