A Case for Automatic Exception Handling

Bruno Cabral, Paulo Marques
CISUC, University of Coimbra, Portugal
{bcabral, pmarques}@dei.uc.pt

Abstract

Exception handling mechanisms have been around for more than 30 years. Nevertheless, modern exceptions systems are not very different from the early models. Programming languages designers often neglect the exception mechanism and look at it more like an add-on for their language instead of 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. We propose a new model that automates the handling of exceptions by the runtime platform. This 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.

1. Introduction

The large majority of modern programming languages rely on exception handling constructs for dealing with errors and abnormal situations. Exceptions eliminate the semipredicate problem, give the programmer an efficient error notification mechanism, 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. Nevertheless, recent surveys [1] show that programmers are not using exception handling constructs as a recovery mechanism. Most of time, when an error occurs, exceptions are silenced or just used to terminate a program in an orderly fashion, not really to recover.

Currently, at the root of the program, is how the mechanisms are used. In most cases, strategies to deal with exceptions are non-existent or 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. 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% (independently of the programming language and exception model) [1]. 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.

The unwillingness of software designers to deal with exceptions correctly and follow some well known best-practices for exception handling [2] will, undoubtedly, contribute to the lowering of the quality of the programs' code and its resilience to errors. It is obvious that something is not right with current exception handling models: they are not adequate enough for developers.

We believe that a new exception model is necessary; one that provides effective exception handling and does 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 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 free from the "burden" of writing exception handling code for a large number of situations.

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. Automatic exception handling should work as a GC for exceptions in the sense that, without (or with minimal) programmer intervention, the mechanism should automatically execute 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.

2. Motivation

The programmer's 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, some authors argue that the mechanism itself can be seriously flawed. For instance, Garcia et al. [3] 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 exceptions models make the writing of exception handling code with quality a cumbersome task. To correctly handle errors or any kind of exceptional situation, the code catching the exception must be able to identify the problem without any margin for uncertainty. For doing so, developers have to choose the correct class to instantiate for the exception being raised. The object representing the exception has to be filled with useful data allowing the "catching code" to provide a valid handling action or even to recover from the error.

In our experience [1], we have seen that programmers raise specific types of exceptions but fail to provide handlers that target those exceptions in detail. I.e. the exception handling blocks, in most cases, are prepared to handle generic exception types and not the specific types that better represent the abnormal situation at hands. This practice is the first evidence that something is wrong.

The Java code block in Figure 1 is a small portion of the source code of the *LimeWire* client for the *Gnutella peer-to-peer* network. This code is responsible for the sending of UDP packets in the application.

```
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);
}
```

Figure 1 – Sample from LimeWire

This 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.

Exception documentation also plays an important role in the quality of an application's code. 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. (After all, if one wants to develop robust software, exception handling is necessary.) Checked exceptions allow the programmer to discover which exceptions may be raised by a method at compile-time but, the unchecked models do not provide such functionality. Thus the

programmer has to trust on the documentation to ensure that he is handling every possible exception. Unfortunately, exception documentation in most programs and software libraries is of poor quality or completely inexistent [4]. Some development platforms have implemented mechanisms, on compilers and IDEs, to inform the developer of possible problems on the code. Such initiatives are truly welcome and are the proof that some automation is needed even for the identification of potential problems.

The automation of exception handling, even if it is applicable only to simple exceptions and runtime exceptions, decreases the amount of code that the programmer effectively writes, increases code quality, speeds up testing procedures/development time, and, at the same time, eliminates some common exception handling bad-practices, such as:

1. The use of empty handlers and silencing of exceptions.
2. The use of general handling code for very distinct exception types.
3. The duplication of code through the application.
4. The excessive mingling of business code with exceptions handling code.
5. The lack of testing of the handlers code;
6. The existence of unhandled exceptions.

Quoting Garcia et al. [3] - *"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 be automatic exception handling for the most common cases.

3. Automatic Exception Handling Model

In this section we discuss the architecture of an automatic exception handling system.

The key point to the success of the automated recovery system 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.

At the platform level, applications have available presets of benign recovery actions to be executed in the event of an exception occurrence. 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 an ACID 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. These actions must not affect the environment on which the program is executing in a harmful or potentially destructive way.

In terms of programming model, a system for automatically handling exceptions could work as follows:

1. The programmer writes try statements for blocks of code where exceptions can be raised. These are called "protected blocks". By default, try statements do not have catch blocks.
2. To explicitly handle an exception, the programmer has to provide a catch block.
3. If no catch block is provided, automatic exception handling should be assured by the runtime for the corresponding try block.
4. To deploy the application, the runtime system requires a configuration file that specifies what happens for each exception type across the program or even for each specific try block. Possible actions include: i) execute a number of recovery actions before throwing an exception in case of failing to recover; ii) directly throwing the exception assuming that it will be caught at a higher level in the stack, or eventually abort the program.

In terms of the runtime, when an exception occurs, the system tries to execute each of the configured recovery blocks, one-at-the-time. After each recovery block is executed, the try block is re-executed from the beginning (similar to a retry action). This happens until either the execution succeeds or all options are exhausted. At that point, according to the 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 must be transactional [5]. 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.

4. Model Evaluation

Due to space restrictions we are not able to provide a detailed description of the prototype implementation for the automatic exception handling model. But, in broader terms, the framework was composed by:

1. A modified Java compiler which makes catch blocks non-compulsory.

2. A custom-made Software Transactional Memory (STM) system implemented as a library.
3. A Java system class loader which performs bytecode instrumentation at load time. That instrumentation allows invoking the correct recovery blocks for each transactional `try` block, according to the application deployment configuration file.

The framework was used to validate the proposed model. It allowed the execution of applications using the automatic exception handling mechanism, providing the necessary means to establish comparisons with other versions of the same applications executing on an unmodified Java runtime.

To evaluate the effectiveness of the approach we used 11 different applications. These consisted in the examples that come with the *Sun's Community Version of its Java System Message Queue* (MQ) framework – *GlassFish*, 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.

Three 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.

Regarding the first metric, the usage of system level automatic recovery code (applied to only one exception type - JMSException) allowed the elimination of 143 exception handling blocks from the original programs, representing 25% of all source code dedicated to exception handling.

To measure application's resilience, we built a fault injector that systematically raises exceptions inside of `try` blocks, monitoring the subsequent behavior of the applications. We performed two different analyses of results: the first looked for visible errors in the programs output; while the second verified if the applications, in the presence of errors, were able or not to fulfill their objectives. In the first set of tests we observed that by using simple recovery strategies there are less 79% application crashes in the modified applications. The second group of tests reported a success rate 88% higher in the modified programs then in the originals, meaning that applications where able to successfully recover in many more cases.

In our framework implementation, the performance penalty for doing automatic exception handling was negligible.

5. Conclusion

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, or with programmers' lack of commitment with fault tolerance issues.

In this paper we argued that future programming languages and runtime environments should provide for automatic exception handling. Exception handling, whenever viable, should be done automatically at the runtime, as currently happens with memory allocation and garbage collectors.

Our prototype shows that the approach is viable, increasing the resilience of programs to exceptions while, at the same time, freeing the programming from writing exception handling code.

Acknowledgments

This investigation was partially supported by the Portuguese Research Agency – FCT, through a Ph.D. scholarship (SFRH/ BD/ 12549/ 2003), and by CISUC (R&D Unit 326/97).

References

[1] 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.

[2] R. Wirfs-Brock, "Toward Exception-Handling Best Practices and Patterns," in *IEEE Software*, Vol. 23(5), September/October, 2006.

[3] 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.

[4] 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 .

[5] N. Shavit and D. Touitou. "Software Transactional Memory", in *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing (PODC'95)*, ACM Press, New York, NY, USA, 1995.