Making Exception Handling Work

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

ABSTRACT

Most modern programming languages rely on exceptions for dealing with errors. Although exception handling was a significant improvement over other mechanisms like checking return codes, it's far from perfect. In fact, it can be argued that this mechanism is seriously flawed. In this paper we argue that exception handling should be automatically done at the runtime/operating system level. The motivation is similar to the one that lead to garbage collection: memory management was a tedious and error prone process, thus virtual machines included support for taking care of it. We believe that many exceptions can be automatically dealt with, and recovered, as long as appropriate mechanisms exist in the runtime environment. We believe that this approach may dramatically influence the way programming languages are designed and significantly contribute to having more robust code, being actually developed with much less programming effort.

1. INTRODUCTION

Most modern programming languages like C#, Java or Python rely on exceptions for dealing with errors. Although exception handling was a significant improvement over other mechanisms like checking return codes and error flags, it's far from perfect. In fact, it can be argued that this mechanism is seriously flawed. For instance, programmers mostly throw generic exceptions which prevent proper handling of errors and recovery for abnormal situations without shutting down the application; programmers catch generic exceptions, not proving proper error handling; programmers that try to provide proper exception handling see their productivity seriously impaired. These practices lead to a decrease in software quality and dependability. It is clear that in order to develop high-quality robust software, in a highly productive way, new advances are needed.

We argue that, in general, exception handling should become a platform issue (at the operating system or virtual machine level), involving little intervention from the programmer. Whenever possible, the execution environment should provide the means for automatically trying to recover the system without having to resort to code explicitly written in exception handling blocks.

2. THE APPROACH

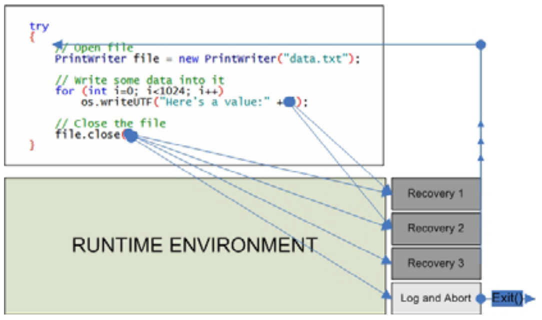
For understanding why we believe that exception handling should become an execution platform level issue, let's consider an analogy with memory management and garbage collectors. In the past, in languages like C and C++, programmers were forced to deal with memory management and the explicit allocation and de-allocation of memory blocks. Although apparently simple, memory management was so error prone that garbage collectors were developed. Garbage collectors provided a way for programmers to stop worrying about memory de-allocation, dangly references, not only paving the way to less buggy software as to improved programming productivity.

Our thesis is that most exception handling should be done automatically by having support on part of the virtual machine and not explicitly by the programmer. The idea is that virtual machines (or operating systems) should have a kind of benign "garbage collector" for exception handling.

One of our main goals it that the programmer, in general, doesn't have to write `catch` blocks. Common exceptions should be handled automatically if they occur. The programmer should only have to mark blocks of code with a simple `try {...}` construct, signaling that an exception may occur in a block and, if it does happen, recovery actions should be taken. Note that including code in `try` blocks does not impair programmer's productivity – thinking about error handling and writing the corresponding code does.

In order to try different recovery strategies when an exception occurs, it is necessary to be able to retry to execute a `try` block (*resumable exceptions*). At the same time, since a `try` block may be executed multiple times while trying to recover from an exception, a "mini-transactional system" has to exist so that `try` blocks start from exactly the same state if executed several times. It should be noted that trying out different recovery schemes when an error occurs is a well-known technique from fault-tolerance: they are called recovery blocks [1], supported by atomic actions. Unfortunately, the technique was never very successful because it relies on writing acceptance tests which detect the occurrence of errors and trigger the recovery mechanism. Practice dictates that writing those acceptance tests is extremely difficult [2] except for a handful of domain applications (e.g. matrix calculations, finite state transition systems, etc.). The interesting aspect of our approach is that acceptance tests (or best said: *error detection*) are given for free since whenever an exception occurs it is clear that the system is not in an acceptable state. In that case, the recovery block can be executed. The transactional system provides for releasing the programmer from having to write state cleaning procedures, which is a quite difficult task to do for generic code.

Figure 1 illustrates the process. The runtime environment provides a set of recovery blocks that should be executed in the presence of an exceptional event, after the execution of a recovery block the targeted `try` block must be re-executed. If the problem persists (the exception still manifests itself) another recovery block should be tried and the process repeated until all the code executes correctly or the application is aborted.



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

    // Write some data into it
    for (int i=0; i<1024; i++)
        os.writeUTF("Here's a value:" + );

    // Close the file
    file.close
}
```

Figure 1. Automatic Exception Handling

Obviously, to allow a correct treatment of an exception inside the recovery blocks contextual information must be available regarding the cause of the fault (e.g. the name of the file that can't be found, the disk identification for the volume that is full, the database connection string regarding the database to which the connection was lost, etc.). This contextual information can be references to faulting components, variables or objects, and are passed to the recovery block as fields of the exception instance.

In this system, the programmer is not completely freed from writing exception handling code. Specific application exceptions still have to be dealt with, and also exceptions for which no general recovery strategy exists. Nevertheless, this system has several important advantages over the existing art: programmers do not have to deal with exception handling in many common cases; the programmer is not introducing bugs by mishandling exceptions or by simply silencing them; when no appropriate reusable recovery blocks exist, the traditional `try-catch` approach is still supported.

3. RELATED WORK

It's been more than three decades since exception handling mechanisms [3, 4] have been around. During this time there have been a growing number of proposals for new ways to detect and handle exceptions. Most of these proposals were strictly attached to the design of new programming languages or programming models. Garcia *et al.* presents a detailed comparison between the different models available in "*A Comparative Study of Exception Handling Mechanisms for Building Dependable Object-Oriented Software*" [5].

Our main contribution comes from combining the idea of exception handling with the one of recovery blocks. In 1974 Horning described recovery blocks as a "program structure for error detection and recovery" [1].

4. CONCLUSION

In this paper we argued that exception handling should be done automatically at the runtime or operating system level, as currently happens with memory allocation and garbage collectors. In our approach, the main difficulty of using recovery blocks, the writing of a proper acceptance test, is eliminated increasing the usefulness of the mechanism. This approach also diminishes the mingling of business code and error handling code. The programmer, in general, doesn't have to think about error handling at the same time it thinks about business code.

The inclusion of a mini-transactional system, besides the obvious benefits in the re-execution of problematic code after the realization of environmental changes, provides the means to incorporate a new retry semantic into platforms that lack from it like Java and .NET.

5. ACKNOWLEDGMENTS

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

6. REFERENCES

[1] J.J. Horning, H.C. Lauer, P.M. Melliar-Smith and B. Randell. A Program Structure for Error Detection and Recovery. In Proceedings of Conference on Operating Systems, IRIA, 1974, 177-193.

[2] B. Randell. System Structure for Software Fault Tolerance. In IEEE Transactions on Software Engineering, 1, 1, June 1975, 220-232.

[3] J. B. Goodenough. Exception handling: issues and a proposed notation. In Communications of the ACM, 18, 12 (December 1975), ACM Press.

[4] F. Cristian. Exception Handling and Software Fault Tolerance. In Proceedings of FTCS-25, 3, IEEE, 1996 (reprinted from FTCS-IO 1980, 97-103).

[5] A. Garcia, C. Rubira, A. Romanovsky, and J. Xu. A Comparative Study of Exception Handling Mechanisms for Building Dependable Object-Oriented Software. In Journal of Systems and Software, 2, November 2001, 197-222