From Errors Treatment to Exceptions Treatment Regarding the Execution Control over Visual Basic Programs

Gheorghe POPESCU, Veronica Adriana POPESCU
Academy of economic Studies, Bucharest
Cristina Raluca POPESCU, University of Bucharest

In order to comply with the quality standards and with the best practices, the execution of the professional programs must be rigorously controlled so that to avoid occurrence of unpredictable situations that might generate anomalies and could lead to computer blockage, forced termination of the execution and data loss.
In traditional programming languages, including Visual Basic 6, the concept of error is extremely evolved. It is considered as error any situation in which the program fails to execute correctly, regardless if such anomaly is generated by a software or hardware cause. Nowadays the modern platforms, including VB.NET have introduced a new concept: exception. Unfortunately, perhaps by mistake, exception is assimilated by many IT specialists as an exceptional (extraordinary) situation or a rare situation.
We agree with the opinion of those IT specialists asserting that error is strictly dependant on the programmer, when he/she fails in correctly generating the application's structures, whilst exception is a situation not fitting in the usual natural execution or as desired by the programmer or user, without meaning that it occurs more often or more rarely.
Designing robust programs implies for such not to terminate abnormally or block, not even upon receiving improper parameters. Two aspects are referred to: the behavior regarding low level errors (caused by the operation system, memory allocation, reading/writing in files or hardware malfunctions) and the reaction to the user's errors, such as providing incorrect input data or incorrect use of operations in respect with their sequences.
Notwithstanding what platform is used in designing the programs and regardless the controversy between the specialists, in order for the execution to be terminated under the program's control, the commands that might generate anomalies and interruptions should be strictly monitored. Implicitly, the execution control should be handed over to specialized procedures, able to analyze the causes generating the anomaly and to launch into execution the most adequate remedy processes, or, as a last resort, to allow for the controlled termination of the program.
These elements that are specific to Visual Basic programming are unitarily and systematically presented both by the traditional unstructured approach of errors and by the modern vision of structured exceptions' approach, finally emphasizing the uncontestable advantages of the .NET platform.
***Keywords*:** *Error Treatment, Exception Treatment, Programming, Program Control, Program Quality, Robustness.*

Introduction

In the era of visual programming, when pseudo-programmers are used so simply move their hands, it is surprisingly to notice that in designing professional programs, which implies: data base design, overlapping layers architectures, objectives focused programming, capitalization of various protocols categories, network security, firewalls and so on, over 60% of the code is dedicated to the errors treatment. Including such code, the concern to the details of errors treatment is what makes the difference between a solid, professional program and a fragile one, which blocks when something unexpected happens. A good rule in programs design is that if something is likely to happen then it will definitely happen, and that if a user can introduce wrong data, regardless how different they are from the real data or how improbable they might be, then he/she will.

The minimal quality prerequisites regarding the safe programs exploiting imply:

- That the program doesn't block, regardless the situation;
- If it is possible to remediate the situation having generated the interruption, such

should be solved in order for the program to continue;
- If the program cannot continue, an adequate explicit message should be displayed, indicating the reason for which the program ends forcedly;
- No data losses should exist upon the abnormal termination of the program. The program should end under control, allowing for the data and the application status to be saved;
- There should be saved, displayed and/or journalized the information necessary in order to find and solve the anomaly (the status having generated the natural interruption of the processing).

In order to achieve these aims, before being implemented programs should be tested, with test data or with real test data, phase which is also known under the name of program mending and fixing (Oprea, 1999).

The main purpose of mending is to eliminate errors from the program. A program's errors can be: syntactical errors, detected upon compiling; connections editing errors (unsatisfied references); execution errors; logical errors.

For the first three mentioned errors categories, the computer (the programming or development environment) provides the programmer with adequate assistance tools (finding and locating).

Logical errors are the most difficult to find, because they are not emphasized and, even though by program execution some results are obtained, such do not guarantee program correctness. Incorrect results can be obtained which can seem correct, or most times correct results can be obtained and only upon certain situations, incorrect results. A program can comprise logical mistakes. In order to eliminate (actually, to limit) them, the program should be tested.

Testing is usually made by specialized personnel, coordinating the entire activity. The number of the testing personnel in the case of complex systems often exceeds by far twice the number of design and creation personnel. The operation supervisors handle the design of tests plans, and also set out the standards for testing, assessing and communicating the results.

Mosley (Mosley, 1993) describes seven categories of application tests. Testing is differentiated depending on the performance manner: manual (made under human control) or automatic (made under computer control) and on the used techniques: static (the source program is verified without being run) or dynamical (the source program is run). Manual tests include: examinations, test run and office verification, whilst automatic tests include: syntactical validation, components testing, integrity testing and system testing. Where:
- Examinations are activities aiming to ensure visual identification in the source code of the most frequent errors, grounded on errors typologies that are specific to each language.
- Test run examines and visualizes the effect of each command in order to identify potential errors (omissions or inconsistencies) depending on the role of the assessor, who can be: the user, the future functional supervisor of the product, the product provider, etc.
- Office verification is the action by means of which a language specialist, usually different from the programmer, verifies the program line by line by pencil and paper. Memory areas are defined, initialized, calculations and verifications are made, exactly as the computer would do.
- Syntactical validation is the only static automatic verification technique performed by a compiler which emphasizes syntax errors.
- Components' testing is the test made over the basic components: functions, procedures, objects or modules, depending on the specific of the language, in order to verify their correct functioning.
- Integrity testing involves testing the assemblies, related modules, in order to verify the inter-operability between their components.
- System testing aims to verify programs as a whole.

1. About Error or Exception in Modern Programming

In traditional programming languages, including Visual Basic 6, the concept of error is extremely evolved. It is considered as error any situation in which the program fails to execute correctly, regardless if such anomaly is generated by a software or hardware cause. Nowadays the modern platforms, including VB.NET have introduced a new concept: exception. Unfortunately, perhaps by mistake, exception is assimilated by many IT specialists as an exceptional (extraordinary) situation or a rare situation.

We agree with the opinion of those IT specialists (Dospinescu, 2004) asserting that **error** is strictly dependant on the programmer, when he/she fails in correctly generating the application's structures, whilst **exception** is a situation not fitting in the usual natural execution or as desired by the programmer or user, without meaning that it occurs more often or more rarely.

Rumbaugh identifies three important quality criteria in writing programs, regardless their complexity (Rumbaugh et al., 1991): code reusing probability, extensibility and robustness.

Robustness implies that a program does not end abnormally not even if it receives inadequate parameters. It refers to two aspects (Ionita, 2003): the behavior regarding low level errors (caused by the operation system, memory allocation, reading/writing in files or hardware malfunctions) and the reaction to the user's errors, such as providing incorrect input data or incorrect use of operations in respect with their sequences.

Most times errors in the first category call only for an elegant interruption of the program, in order to leave the working environment as clean as possible and to save as many information as possible for diagnosis purposes. Robustness upon using errors is however never sacrificed; these sorts of anomalies must be foreseen upon analysis and mentioned in creation specifications, for the purpose of being subsequently solved.

Under such circumstances, the exception is the error situation or the unexpected behavior encountered in a running program. Exceptions can occur both in the running program and in its running environment. Program developers mostly handle program exceptions that can cause its blockage. A program can run perfectly, yet when attempting to save a file the disk is unexpectedly full or the floppy disk unit is not ready, or the floppy disk is not inserted.

However, regardless the specialists' conceptual disputes, professional (industrial) programs should answer to some *minimal quality standards* regarding their safe exploiting. Quality beings by complex understanding of what the program does and the way in which the user will interact with it. The professional designer's objective is to anticipate each possible way in which the program could go wrong, and to take measures in order to fix exceptions. The programmer must completely understand the way in which each variable will interact with the program, the way in which it is stored in CPU registers, the way in which it is processed by the memory and so on.

For the common language routine (CLR) an exception is an object deriving from the System.Exception class, basic class from which all exceptions are inherited. Exception is generated by the code sequence where the problem occurs towards that part of the code which is destined to receive and to treat the error. The type of exception will determine the code which is to treat the exception.

The Visual Basic development environment finds all design errors (syntax errors) occurring when a command is written incorrectly (incorrect key words, missing parameters, incorrect expressions, etc.), providing necessary assistance upon their repairing (placement on the error line, contextual help, assistance in introducing expressions, methods, properties or reserved words, etc.).

Repairing the other types of errors, running and logical, is in charge of the programmer.

In order to test programs, good practices recommend (Harold, 2003):

- For the testing process to be performed rigorously and with discipline;
- For more imagination and perseverance to be used in insisting upon simulating anoma-

lies, because in real life anything that can go wrong will go wrong;
- For the testing process to be carefully tracked, potentially by using a program designed for other programs' testing process;
- For special attention to be paid to testing variables' values;
- For extreme values to be taken into consideration (those at the higher and lower limits of possible variables' values intervals);
- For potential counting errors to be carefully evaluated when the programs has a logical error comprising a repetitive structure;
- For potential irregularities to be tracked down, for which reason the program should be tested in the most diverse situations possible.

Despite the efforts for testing programs even from the programming and implementation phases, in order for such to be safely exploited, there must be included in the program, in its source, in the delicate risk areas, some special procedures for tracking down and treating the anomalies occurring during programs' exploiting.

Detect and repair procedures for the anomalies occurred in programs' exploiting are different in the traditional programming languages compared to the modern platforms. In order to emphasize the basic mechanisms of two of the most common practices, we will describe errors treatment in Visual Basic 6 and exceptions treatment in VB.NET.

2. Unstructured Errors Treatment

The standard method for treating detectable errors in Visual Basic 6 (Popescu, 2005) implies the use of the instruction On Error GoTo label.

Actually, when Visual Basic encounters a detectable error, it interrupts the program running (its natural execution) and transfers the control to the first command in the block identified by label. The command On Error GoTo should be placed before the instructions to the errors of which it refers (in order to configure the errors treatment routine), and before the label a procedure ending (termination) command should be included if the processing ends without an error (in order to not force errors treatment commands' execution even in the case when such errors do not occur).

At the end of errors treatment instructions block the instruction Resume can be placed, which allows for the operation (instruction) which produced the error to be reinitiated. Using this command is benefic only if during error treatment it is possible for such to be fixed, otherwise (if the error persists) Resume instruction can lead to infinite cycling. In order to transfer the control to the first instruction following the one which causes the error (and hence in order to avoid cycling) the Resume Next command should be used, and in order to transfer the control to another line or label within that procedure, the Resume label command must be used (see figure 1).

Errors treatment involves displaying the error number and description and suggestion of several alternatives for Retry, Ignore, or Abandon. The error number is given by the ERR ( ) function. The explanatory message associated to an error number will be transmitted by the ERROR$ (error_number) function. The suggestion regarding processing continuation is made by means of the command buttons within an MsgBox. The code attached to the command buttons can be: Resume (for Retry), Resume Next (for Ignore), Exit Sub (for Cancel) and Resume label (for Resume from a certain label).

A professional approach of errors treatment implies the creation of a customized errors treatment procedure. This procedure should accept as a parameter the number of the error, returned by function ERR ( ) and, depending on the error number, it must display the explanatory message in Romanian language, with the help of a Select Case structure, by using the exhaustive translated list of the traceable errors. It is obvious that in order to make this procedure available to the entire application, it should be defined in a standard module.

The unstructured errors treatment by using On Error statements can really damage the application's performances and can create a difficult repair and maintenance code. First

of all, because we are forced to use the GO TO command, even though under a masked form, and secondly because the unstructured errors treatment mode can be confuse when successive appeals occur of other procedures, potentially written in other programming languages, which include or not their own errors treatment mechanisms.

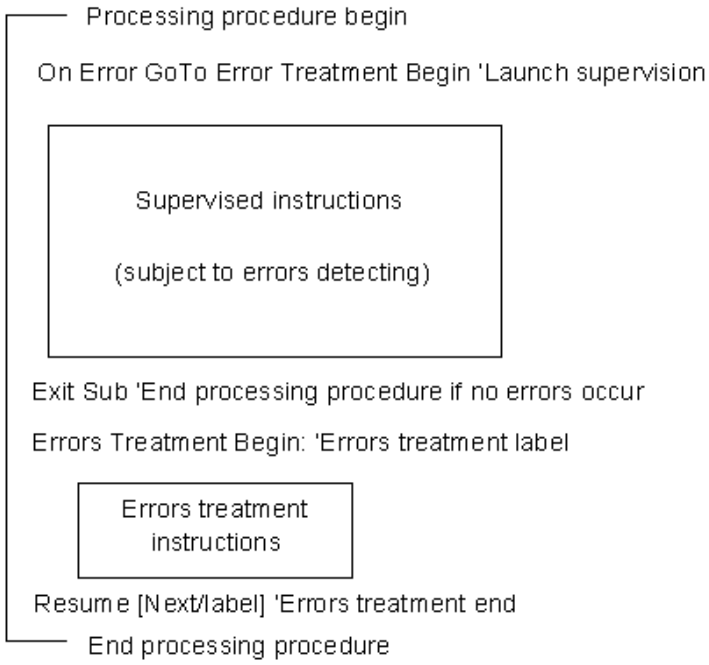

Fig.1. Structure of procedures subjected to errors treatment

3. Structured Exceptions Treatment

In VB.NET systematic exceptions treatment uses an improved version of the Try … Catch … Finally syntax, already supported by other programming languages (Conell, 2002). This form of exceptions treatment combines a modern control structure (similar with Select Case or While), with the exceptions, protected and filtered code blocks:

```
Try  ’ Launch supervision
       Program instructions
       …
       [Catch1 [exception [As type1]] ’ Explicit treatment of the type1 exception
              [When expression]
              …
       [CatchN [exception[As typen]] ’ Explicit treatment of the type n exception
              [When expression]
              …
       [Finally
              Final program instructions]

End Try ’ End of exceptions treatment
```

Where:

- The code in the Try – Catch block represents the code of the program monitored for detecting and repairing potential anomalies;
- The code included in each of the Catch blocks is run in order to treat specific exceptions;
- The code in the Finally code block is executed upon processing termination, notwithstanding if an exception is initiated or not.

In the Catch exception treatment blocks, in order to be sure that unpredictable exceptions will be also tested, it is appropriate for the command Catch except As Exception to be included as a last resort, allowing for the generic treatment of the exceptions which have

not been indicated as specific exceptions in the previous blocks.
The exception generating process is technically referred to as *exception throwing*, whilst its treatment is referred to as *exception catching*.
Actually, if an anomaly occurs that generates an exception in the monitored block in Try area, the natural execution of the program is interrupted (the rest of unexecuted commands are ignored), and the control is transferred to the Catch block. The Catch block receives as argument an instance of the exception which generated the interruption, argument which allows for the type of anomaly to be identified. Depending on the occurred exception, such will be treated in the Catch block, either as a specific anomaly (if it has been explicitly foreseen and treated), or as a generic anomaly, if it hasn't been explicitly foreseen and treated. If the application is complex and calls for final operations: results transfer, files or data base closure, working zone cleaning etc., then these processes will be placed in the Finally block, which will be executed regardless if any exception occurs or not.
For the systematic exceptions treatment in VB.NET, the programmer can use the exception class Exception Class, which provides amongst its properties information regarding the name of the occurred exception, place where the anomaly occurs, the object or application having created the exception, etc. These elements will be valorized in the Catch block in order to identify, explain and/or journalize the found exceptions. All exceptions are deriving from the basic System.Exception class. This class has four main features providing information about the exception (Troelsen, 2002):
- .Message – describes the occurred exception;
- .StackTrace – comprises the set of methods appealed until the moment when the exception occurs.
- If information exist for repair, the name of the file with the source-code and the row number are provided;
- .InnerException – comprises another object Exception. This happens in the case when an exception is cached which is then run through a set of exception processing blocks;
- .HelpLink – is an access way to a Help file which might have more information on the occurred exception.

In order to select the sort of exception to be treated in the Catch block, the programmer benefits of an extensive list provided by the auto fill in system in the Code Editor.

The structured exceptions treatment is an integrated service in the core of the programming framework .NET – in .NET Framework, so that it is available to all languages targeting the .NET platform.
In its intimacy, exceptions treatment implies:
a) Administered execution code of a Visual Basic.NET program meets a Try block;
b) CLR automatically writes a generic exceptions treatment core in the appeals Bulk;
c) CLR appeals the command in the Try block which can generate an exception;
d) The execution code meets an anomaly and generates an exception;
e) CLR looks in the bulk in order to find the potential exception treatment code, locating the appellant;
f) If the code is found, then CLR carries out the bulk;
g) The location information along with the program control are transferred back to the Catch block of the execution program;
h) The Catch block is executed.

The exceptions which are not included in a Catch block are treated by the common language execution routine; however the main concern of a professional programmer should be to avoid at any cost such situations.
From the Exception class there derive a series of other specialized classes which, depending on the needs, can be used in exceptions treatment. Moreover, the programmer can define his own classes deriving from the exceptions treatment system classes for a unitary treatment and solving of the exceptions.

Two of the most important classes deriving from System.Exception are System.Application (for treating errors generated by the application, and not by CLR) and System.SystemException, which is the basic class for all exceptions involved in the name area System.

Conclusions

A mandatory prerequisite for professional programs is to control the program's running by foreseeing any potential situations that might affect the natural carrying out of the processes in both the program and its running environment (Nastase et al., 2007). These should uniformly and elegantly treat the most specific situations of the foreseen anomalies in the relationships with the user and execution environment. The repair will be ensured by displaying adequate messages, specific to the encountered irregularities, and then of the more general situations, in the case of which regardless if they cannot be repaired and impose running interruption, such should be made under control, by displaying an explanatory error message, even by journalizing the conditions in which the irreparable interruption was generated and mandatory by closing the files, saving the data and processes performed up to the moment of the incident.

Not treating the anomalies by means of the program implies that when such occur, the program blocks, and a general encrypted system message is displayed. Program termination would be very unpleasant, implying data loss and sudden processes finalization, upon ambiguity over the status of the program and over the data that most times impose rebooting and re-indexation or worst reinstalling and re-initializations of the processes in previous phases with well defined statuses.

At the present moment the .NET platform includes a very powerful exceptions treatment tool. It can be definitely said that the exceptions treatment mode, which is based on exception objects and protected code blocks has reached its maturity.

References

- Connell, J. (2001) *Microsoft Visual Basic .NET*, Redmond: Microsoft Press
- Dospinescu, O. (2004) *Dezvoltarea aplicatiilor in Visual Basic.*, Iasi: Polirom
- Harold, D. (2003) *Visual Basic .NET for windows,* SUA: Pearson Education, Inc. Publishing as Peachpit Press
- Ionita, A.D. (2003) *Modelarea UML in ingineria sistemelor de programe,* Bucuresti: Bic All
- Mosley, D.J. (1993) *The Handbook of MIS Application Software Testing*, Englewood Cliffs, New Jersey: Yourdon Press
- Oprea, D. (1999) *Analiza si proiectarea sistemelor informationale economice,* Iasi: Polirom
- Popescu, Gh. (2005) *Programarea calculatoarelor in Visual Basic*, Bucuresti: Editura Gestiunea
- Rumbaugh, J. & Blaha, M. & Premeriani, W. & Eddy, F. & Lorensen, W. (1991) *Object-Oriented Modelling and Desing*, SUA: Prentice-Hall
- Troelsen, A., (2002) *Visual Basic.NET and the .NET Platform: An Advanced Guide,* SUA: Apress