Exception Handling versus Fault Tolerance

Jørgen Lindskov Knudsen
Computer Science Department, Aarhus University
Aabogade 34, DK-8200 Aarhus C, Denmark
Tel.: +45 89 42 56 69 – Fax.: +45 89 42 56 24
E-mail: jlknudsen@daimi.au.dk

A large part of any software design is the handling of error situations or rare situations that are unlikely to happen. For small programs, it is common to print an error message and terminate the program execution. The user must then correct the error and execute the program again.

For most non-trivial programs it is not satisfactory just to terminate the program with an error message. Consider the situation where the program is a word processor. An error message saying e.g. that some internal register has overflowed is in general not meaningful to a user of such a word processor. Another example is a flight reservation system. Such a system runs over a very long period of time, and it would not be acceptable if e.g. a table overflow simply resulted in an error message and a subsequent termination of the program.

There are many types of errors or exceptional situations that a program may have to deal with. An *exception* is a class of computational states that requires an extraordinary computation. It is not possible to give a precise definition of when a computational state should be classified as an *exception occurrence*; this is a decision for the programmer. In practice, most people have a good feeling of what is the main computation and what are exceptional situations. The exceptional situations are all those situations that imply that the main computation fails.

A program must be able to deal with exceptions. A good design rule is to list explicitly the situations that may cause a program to *break down*. Many programming languages have special constructs for describing *exception handling.*

Many proposals have been put forward on defining language constructs for handling exceptions. Most of these are dynamic in nature (i.e. the exception is handled by some component found by examining the dynamic calling sequence leading to the exceptional occurrence), while others are static (i.e. the exception is handled by some component found by analyzing the static context of the exception occurrence).

This short paper describes the BETA approach to exception handling and fault tolerance programming, gives a few hints to the differences between exception handling and fault tolerance programming, as well as a few pointers to existing exception handling constructs.

1. STATIC EXCEPTION HANDLING IN BETA

In BETA [Madsen et al. 93], exception handling is introduced into the language in the form of static exception handling [Knudsen84], [Knudsen87]. The support is introduced without actually adding any new language constructs, but only by adding a predefined pattern[1].

A virtual pattern dealing with an exception is called an *exception pattern*, or just an *exception*. The invocation of an exception pattern is called an *exception occurrence*. An exception is *raised* when an exception pattern is invoked. The code associated with an exception pattern is called a *handler* or an *exception handler.*

The code associated with a specialization of the `exception` pattern will be the *default handler* for the exception in the case where no further binding of it is made. A sub-pattern may extend the default handler by a further binding. A specific *instance handler* may be associated with each instance by instantiating a singular object with a further binding of the exception pattern.

1.1 Experiences

The exception handling mechanism shortly introduced above is based on the static approach to exception handling. The benefits of this approach is a very declarative exception handling style, where the consequences of raising an exception can be deduced from the static properties of the program. This also implies that the cost of these mechanisms are extremely low, both in the case of the exceptions never being raised, and in the case of an exception being raised. The cost of static exception handling is fully comparable with ordinary programming.

It is our experience, that static exception handling is the exception handling model per se for well-designed object-oriented systems. It focus on the use-relation between objects: it is the responsibility of the user (or client) of an object (or a service) to specify explicitly the consequences of exception occurrences in the object. The static exception handling model gives the ability for an object (through its pattern declaration) to specify the exception occurrences that may occur within it (by declaring virtual attributes, that are specialization's of the `exception` pattern). Moreover, the static exception handling model offers the ability for the client of the object (or service) to explicitly (and statically) to specify the handling of such exception occurrences through virtual bindings of these exception patterns. The effectiveness of static exception handling is demonstrated by the fact, that the entire Mjølner System[2] is programmed entirely using static exception handling as the only exception handling model.

2. DYNAMIC EXCEPTION HANDLING IN BETA

The static exception handling model is based on two assumptions:

Perfect Design
: There is an underlying assumption that all users of a given pattern with exception specifications makes a complete handling of the exceptional occurrences (i.e. further binds the proper virtual).

[1] Pattern is the BETA name for a class – this is not the whole story, but hopefully sufficient.
[2] You can find more information on the Mjølner System at http://www.mjolner.com/mjolner-system.

Nearly by definition, designers will make mistakes, forgetting to take care of some exceptional occurrences, giving rise to the program being terminated.

Object Creation

There is also an underlying assumption that the objects being used by the application, are also created by that application (making it possible to handle the exceptional occurrences in these objects through static exception handling).

In a persistent or distributed environment, objects are not created only by the running application, but also by other applications and made available to the running application.

This implies that there is a need for offering an additional model for exception handling in order to be able to support truly fault-tolerant programming with respect to exception handling.

Error handling can therefore be divided into two distinct disciplines: exception handling and fault-tolerant programming. Exception handling deals with the handling of well-defined error conditions within a well-defined system or framework. And fault tolerant programming deals with error handling in all other cases: ill-designed systems, faults in the exception handling code, errors originating from outside the system or framework.

Our experience with the static exception handling mechanisms of BETA have proved static error handling as an effective exception handling mechanism, but also that it is difficult (and in some cases impossible) to use for fault tolerant programming.

We therefore propose the introduction of a dynamic error handling mechanism to be used for effective fault tolerant programming.

We would like to stress that the static exception handling model should be used almost exclusively, since it gives the most well-designed exception handling, integrated with the object-oriented tradition, and reserve dynamic exception handling only to those case where no other error handling solution can be found. One could say, that the relation between static and dynamic exception handling is fairly similar to the relation between structured programming and the GOTO controversy.

In the dynamic approach, there is no static connection between the definitions of an exception, the raising of an exception, and the actual handling of an exception. Exceptions are defined anywhere in the program, and may be raised anywhere where these exceptions are visible in the program text. Handling of the exception is on the other hand possible in all parts of the program (even places where the exact definition of the exception is unknown).

2.1 Dynamic Exception Handling Model in BETA

The dynamic model for exception handling for BETA is heavily inspired by the C++ model for exception handling, which in turn is inspired by the ML model for exception handling[3].

The dynamic exception handling model for BETA is concentrated around the following four main concepts: *exception objects*, *throwing exceptions*, *try blocks*, and *exception handlers*.

[3] It should be noted that this is a proposal for extending the BETA exception handling model. The proposal has not been certified, and the Mjølner System does not implement it.

Exception Objects

An exception object is a regular BETA object. The purpose of an exception object is to act as a messenger between the point where the exception occurrence have been identified, and the place where the exception is handled. The exception object may have attributes, carrying information from the exception occurrence to the exception handler, but it may also act merely as a signal (without attributes).

Throwing Exceptions

When an exception occurrence have been identified, the dynamic exception model offers the possibility of *throwing* an exception object. When an exception object is thrown, the intuition is that the exception object 'travels' back the dynamic call-chain until an exception handler for this exception object type have been located. When located, the particular exception handler is given access to the exception object, and may then initiate proper exception handling processing, possibly based on the information brought to it by the exception object. During the processing of the exception object, the exception handler may decide on the proper continuation of execution of the application.

If the entire dynamic call chain have been exhausted in the search for an exception handler and no matching exception handler have been found, then the exception is automatically converted into an instance of the predefined exception *unknown*. This *unknown* exception object contains a reference to the original exception object and is automatically thrown at the same spot as the original exception object. If any handlers on the dynamic call chain do also not handle this unknown exception object, the *unknown* exception object is raised as a static exception occurrence, giving raise to termination of the entire application.

Try Blocks

The BETA model for dynamic exception handling is based on *try blocks* as the means for specifying the extent of exception handlers. A try block is a special kind of nested blocks (similar to nested blocks in ALGOL, PASCAL and C/C++). The purpose of a try block is to function as a definition place for exception handlers, and a try block is capable of handling those exceptions for which there are defined a handler. In the description of the semantics of throwing an exception object, it was mentioned that a handler was sought. To be more specific: during a throw of an exception object, the dynamic call-chain is scanned to find the first try block with an exception handler, matching the exception object. If no matching exception handler is found in a try block, the exception is automatically propagated to the next try block in the dynamic call chain.

Exception Handlers

As described above, dynamic exception handlers are defined in try blocks. An exception handler is capable of handling a series of exceptions through the specification of a series of *when-clause*s. Each when-clause is capable of handling one particular exception object type (or any subtype hereof).

The sequence of when-clauses in a handler is important, since more than one when-clause in a handler may match a given exception object (the two subtypes overlap). The handler handles this potential ambiguity by choosing the first when-clause that matches the particular exception object.

During the handling of an exception in a when-clause, the when-clause has access to the exception object being handled.

Execution Control

During the handling of an exception object, the chosen when-clause has five different possibilities for controlling the execution of the program, namely *continue*, *propagate*, *retry*, *abort*, and *terminate*.

Continue:
If continue is chosen, the exception occurrence have been fully recovered, and the execution may continue from the spot, where the exception object was originally thrown.

Propagate:
If propagate is chosen, the exception object is propagated further backwards along the dynamic call chain in order to be further handled by some other try block. Propagation is the default for exception objects for which no exception handlers are found in a try block. Propagation is also the default for exception objects with a matching when-clause if no other execution control is specified in the when-clause. Propagation implies that the exception handling have only partially been concluded.

Retry:
If retry is chosen, the execution is resumed from the beginning of the try block in which the chosen when-clause is specified. Retry implies that the best way to continue the application is to re-execute the entire execution from which the exception occurrence arose. Usually this implies that the exception handling have brought the application back to a stable state.

Abort:
If abort is chosen, the execution is resumed *after* the try block in which the chosen when-clause is specified. Abort implies that the actions of the exception handler have replaced the remained of the try block (i.e. the actions after the spot, where the exception object was thrown).

Terminate:
If terminate is chosen, the execution of the entire application is terminated. Choosing terminate implies that the exception is impossible to handle (i.e. the exception occurrence is indeed severe).

3. A SHORT OVERVIEW OF EXCEPTION HANDLING MECHANISMS

The foundation of most research on exception handling is the pioneering work by J.B. Goodenough [Goodenough75]. Most procedural languages with special language facilities for exception handling are more or less directly based on this work (e.g. languages like Clu and Ada). Several object-oriented languages have included special language facilities for exception handling (e.g. Smalltalk, Eiffel, C++, and Java).

All these facilities employ a dynamic approach to finding the handlers of a particular exception. This implies (with variations) that the handler for an exception is found by traversing the call chain of procedure invocations and enclosed block backwards, until a block or a procedure invocation is found in which a handler for the exception is defined. This dynamic approach implies the separate definition of the exception and the handler, and association of the exception with the handler based on the dynamic behavior of the

program. This implies that it is very difficult to trace the exceptional computation (works somewhat like a series of computed GOTOs) and it is very difficult to ensure that all exception occurrences will be handled eventually (i.e. it is very difficult to verify that a program will respond sensible to all perceived exceptional conditions)[4].

This dynamic behavior (dynamic binding of handlers to exceptions) is often in contrast to the host language (e.g. Ada and Clu) that uses static name binding (e.g. when binding procedure invocations to procedure declarations). This has resulted in criticisms from several sources. E.g. C.A.R. Hoare [Hoare81] states that "... *the objectives of languages including reliability, readability, formality and even simplicity ... have been sacrificed ... by a plethora of features ... many of them unnecessary and some of them, like exception handling, even dangerous.*"

As an alternative to the dynamic approach to exception handling, a proposal has been made for a static approach to exception handling [Knudsen84, Knudsen87] that is based on the sequel concept, proposed by R. Tennent [Tennent77]. The static approach have been further developed to incorporate support for smooth termination (i.e. allowing for clean-up etc. of blocks being terminated during an exceptional "backtrack") [Knudsen87].

The BETA approach to static exception handling is inspired by this static approach to exception handling. The rationale for the concrete design have been to introduce static exception handling into BETA without introducing any new language constructs, but instead by utilizing the powerful abstraction mechanisms of the language to construct an exception handling concept.

Static exceptions in BETA are designed such that ignoring to handle an exception will automatically terminate the entire `program` block. The handler of an exception may be split into different parts, spanning several block levels, resulting in smooth termination of the different blocks. The termination level of static BETA exceptions is default defined to be the `program` block. The default termination level of a static BETA exception can be redefined in the handler. If the programmer wishes to define another termination level, this is done in the `do` part of the handler by specifying `leave B`, where `B` is the label of some enclosing block. The programmer may also choose to restart some part of the computation as a result of the static exception. This is also done in the `do` part of the handler by specifying `restart B` where `B` is the label of some enclosing block. Finally, the programmer may choose to resume the computation immediately after handling the computation. This is done by specifying `continue` in the handler (i.e. in the `do` part of the exception). In order to be able to construct fault tolerant programs, BETA offers a dynamic exception handling mechanism, which e.g. is used to enable coping with static exceptions that are not handled, enabling the construction of fault tolerant programs.

Dynamic exceptions in BETA are very similar to those dynamic exceptions found in languages like C++, Java, and ML. When comparing with those approaches, the BETA dynamic exceptions have been introduced without the need to introduce any new language constructs, and the matching process during the dynamic search for a handler along the call-chain is much more flexible. The BETA dynamic exception matching includes facilities for exception object identity matching and exception information matching – facilities which, to our knowledge, cannot be found in other dynamic exception models.

[4] It is important to note that exception handling in any programming language is only capable of handling exceptional conditions that have been perceived during program design.

4. EXCEPTION HANDLING VERSUS FAULT TOLERANCE

As it can de seen above, it might be too simple to deal with the problem of errors in programming from one perspective, namely the predominant view of dynamic exception handling. There seem to be two different, but closely related problems here, namely that of exception handling and that of fault tolerant programming.

Let us try to define these two concepts:

Exception handling:

Exception handling is the techniques by which the designer of a piece of software – an abstraction (library, module, class, etc.) can define possible exceptional occurrences that are expected to occur in the abstraction. Moreover, these exceptional occurrences are part of the definition of the abstraction in the sense that the users of the abstraction knows about the possibilities of these exceptional occurrences, and thereby are able (or forced) to deal with these occurrences when using the abstraction.

Fault tolerant programming:

Fault tolerant programming, on the other hand, is the techniques to cope with exceptional occurrences, that are not defined as part of the abstraction definition. Examples of such occurrences are programming errors in the code of the abstraction, unhandled exceptions, and unexpected system states (e.g. the disk system suddently becomes inaccessible).

The experiences with the BETA exception handling mechanism indicates the need for separation these two issues, possibly giving rise to two separate, but related language mechanisms for exception handling and fault tolerant programming.

Interestingly, the dynamic trend (exemplified by Java) is to introduce more static analysis in order to make exception handling more safe (exceptions are declared in the interfaces, and the language compiler enforces static checks to ensure, that these exceptions are handled). This gives a static verifiability of an otherwise dynamic mechanism. On the other hand, the static constructs have realized that there are cases, where the static nature of the constructs makes it very difficult to ensure that a program never terminates due to an unhandled exception.

Looking more closely at the Java exception handling mechanism [Java], one will find an interesting change in the rules of the game when we investigate the rules concerning the very basic exceptions (such as numeric exceptions). While the compiler enforces the handling of all exceptions in an interface, the compiler does not enforce this rule on the basic exceptions[5].

Looking more closely into the arguments for this, it is interesting to see, that sentences like “it would become tedious top properly declare them all, since practically any method can conceivably generate them” are given. And an example is “every method running on a buggy Java interpreter can throw an `InternalError` exception”.

One interpretation of these differences in semantics is exactly, that regular exceptions (i.e. Throwables) are handled using *exception handling*, whereas non-Throwables are handled using *fault tolerance.* In the Java case, this implies that the type system of exception are used to differentiate between exception handling and fault tolerance.

[5] To be specific, the Java compiler only enforces this rule on exceptions that are instances of subclasses of `Throwable`.

In the BETA case, the type system of exceptions does not introduce any differentiation between exceptions for exception handling and exceptions for fault tolerance. This is actually a deliberate design decision. The reason is, that in some parts of a system, a given exceptional case might be known as a part of the definition of an abstraction, whereas the same exceptional case may occur other places, where it is unreasonable to define it as part of the abstraction. In the Java case, this situation imply that the system must have two exception types defined – one for the exception handling case, and another for the fault tolerance case. In BETA, this is handled by the same exception definition. If the exception is raised using the static exception handling system, then the exception is handled using *exception handling*, whereas if the exception is raised using the dynamic exception handling mechanisms, then the exception is handled using *fault tolerance*.

It should be noted, that the BETA mechanisms are actually connected, such that if an exception is raised as an static exception, but unfortunately not handled by anyone, then the exception ahndling system will automatically convert it into a dynamic exception, implying that the error is converted to a fault to be handled through *fault tolerance*.

In conclusion, it is the current understanding of this author, that the relationship between exception handling and fault tolerance needs further investigation, and that there is a need for further development in the area og language constructs for supporting both exception handling and fault tolerance. However, it seems important not to do this through separate language constructs, but the develop interconnected language constructs, enabling there two approaches to error handling to be integrated, improving the stability of future system designs.

5. REFERENCES

[Eiffel89] Eiffel: The Language, (Version 2.2), Interactive Software Engineering Inc., Santa Barbara, CA, USA, 1989

[Ellis90] Margaret A. Ellis and Bjarne Stroustrup, The Annotated C++ Reference Manual, Addison-Wesley, 1990

[Goodenough75] J.B. Goodenough, Exception Handling: Issues and a Proposed Notion, Comm. ACM, 18(12), Dec 1975, (683–696)

[Hoare81] C.A.R. Hoare, The Emperor's Old Clothes, Comm. ACM, 24(2), Feb 1981, (75–83)

[Java] David Flanagan, Java in a Nutshell, Second Edition, O'Reilly.

[Knudsen84] Jørgen Lindskov Knudsen, Exception Handling — A Static Approach, Software, Practice and Excerience, 14(5), May 1984, (429–449)

[Knudsen87] Jørgen Lindskov Knudsen, Better Exception Handling in Block-Structured Systems, IEEE Software, 4(3), May 1987

[Madsen et al. 93] O.L. Madsen, B.Møller-Pedersen, and K. Nygaard, Object-Oriented Programming in the BETA Programming Language, Addison Wesley, Reading, MA, June 1993.

[Tennent77] R.D. Tennent, Language Design Methods based on Semantic Principles, Acta Informatica, 8(2), 1977, (97–112)

[Wikstrom87] A. Wikstrøm, Functional Programming Using Standard ML, Prentice-Hall Inc., 1987