PUC

Monografias em Ciência da Computação
nº 23/92

Exception Handling in Object Oriented Languages: a Proposal

Sérgio E. R. Carvalho

Departamento de Informática

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO
RUA MARQUÊS DE SÃO VICENTE, 225 - CEP 22454-970
RIO DE JANEIRO - BRASIL

PUC RIO - DEPARTAMENTO DE INFORMÁTICA

Monografias em Ciência da Computação, Nº 23/92
Editor: Carlos J. P. Lucena Julho, 1992

Exception Handling in Object Oriented Languages: a Proposal *

Sérgio E. R. Carvalho

* This work has been sponsored by the Secretaria de Ciência e Tecnologia da Presidência da República Federativa do Brasil.

EXCEPTION HANDLING IN OBJECT ORIENTED LANGUAGES: A PROPOSAL

Sergio Carvalho
Departamento de Informática, Pontificia Universidade Católica, Riode Janeiro.

Resumo

Um mecanismo para o tratamento de exceções em linguagens orientadas a objetos é apresentado. Inicialmente o modelo de linguagem adotado é descrito, particularmente o conceito de "classe". O tratamento proposto combina com naturalidade a semântica de classes escolhida com a semântica de exceções adotada em algumas linguagens de programação convencionais. A familiaridade obtida é importante fator na aceitação de orientação a objetos por programadores convencionais.

Abstract

A mechanism for exception handling in object oriented languages is proposed. Initially the language model chosen is described, particularly its "class" concept. The proposed mechanism combines naturally the semantics of classes with the semantics of exceptions in conventional languages. The resulting familiarity is important to conventional programmers trying to embrace object orientation.

Keywords

object orientation, exception handling, class.

EXCEPTION HANDLING IN OBJECT ORIENTED LANGUAGES: A PROPOSAL

1. Introduction

The object oriented approach for the construction of software systems is becoming increasingly popular. Correspondingly, pure object oriented languages are being designed and implemented, as for example Eiffel [Meyer 88] and Trellis [Schaffert 86]. Also, old conventional languages, for example C and Pascal, are being (inelegantly ?) adapted to accomodate object orientation. Whatever the case may be, the class concept is central to the design of object oriented languages. Several interpretations can be given to this concept. Once a class interpretation is chosen, other features can be designed to fit: inheritance, polymorphism, and exception handling, for example. In section 2 we present a view of classes. This view strongly underlines the design of the exception handling mechanism described in section 3. In section 4 we present our conclusions.

2. A View of Classes

For the purposes of this report, we adopt Wegner's view that a class is basically an implementation of an abstract data type affected by inheritance [Wegner 86]. In more detail:

- a class is a passive syntactic structure encapsulating the definition of both the data structure and the operations applicable to the instances (objects) it models;
- a new class may be derived from (may inherit properties of) other already existing classes.

This "static" or "passive" view of classes is extremely convenient for the popularity and ease of use of new object oriented languages: classes can thus be viewed as mere extensions of types, a familiar concept to conventional language programmers. Another advantage of classes as static structures modelling objects is the strong typing that can be obtained: language compilers can generate efficient, optimized code by discovering, each time a strongly bound object is used, to which class it belongs. Also, in this view,

classes can be designed, compiled and tested separately, and can then be stored in libraries for further reuse.

To consolidate this view, the following skeletal definition of class is suggested:

```
class class-name
        inherits class-name,...
        exports fields,... operations,...
        imports class-name,...
        structure
                field,...
        behavior
                operation,...
end class
```

The **inherits**, **exports** and **imports** clauses constitute the class interface. The **exports** clause lists the fields and operations visible outside the class. In the **imports** clause we declare all external classes used to model, for example, structure fields or local operation objects.

In the **structure** section, we define the field composition necessary to describe the state of a class object. This composition is like that found in conventional language records, but here inheritance also plays a part: the data structure for objects of class A is found by adding to the fields declared in class A the fields found in the declarations of all superclasses of A, listed in class A's inheritance clause, and so transitively on. This is usually accompanied by disambiguating rules, applied when a same name field appears in more than one class in this inheritance chain.

Access to object fields is in general restricted to the class operations defined in the behavior section (below). In this way an abstract view of the class is presented to users, who can then affect the state of an object solely through class operations.

In the **behavior** section we declare all operations applicable to the modelled objects. Typically, the semantic of these operations is either that of straight procedure calls, as in Pascal, or that of remote procedure calls, where for example the operation to be executed next is chosen according to some queuing policy.

In fact, operations of different natures may be needed in the behavior sections of class declarations. This is the case of object oriented languages designed to support the construction of graphical user interface applications.

where the user may asynchronously interfere with the execution by, for example, pressing a button, or by selecting a menu option. On the graphical development platforms existing today (Windows for DOS and MOTIF for UNIX, for example), one such asynchronous action causes a message to be queued up, and, eventually, causes the execution of some associated operation to implement the user's wishes.

To represent operations of different natures in the behavior section of class declarations, different active units should be available in the language. A few can be named:

- procedure-like units, seizing control when called, and relinquishing it upon termination;
- message handling units, invoked when the corresponding message is removed from the system's queue;
- iterators (as in CLU), designed to provide, one at a time, a series of values for a given object, typically to control a loop statement.

The above units differ in such aspects as local environment life cycle, entry points and return addresses. Local environments are created and deleted for procedure like units and for message handlers; iterator environments are retained between consecutive value producing cycles. Return addresses are kept for procedure-like units, but not for message handlers; for iterators, both a yielding address (where the value produced is used) and a resume address (where the execution of the iterator continues) must be preserved.

Operations of all kinds are applicable to receiving objects of the corresponding class, actually special parameters to the operation.

Besides modelling the state and behavior of objects, as shown above, classes may contain other convenient features. As an example, one may consider the existence of class environments, actually static data structures visible to all class operations, and belonging to the class itself, not to each of its objects.

The syntactic and semantic class structures described above suffice to model the exception handling features proposed in the next section.

---

3. Exception Handling and Object Orientation

In section 2 above we presented a short description of a convenient class mechanism. In this section we discuss exception handling for that particular model. We do not propose to discuss exception handling in detail; for this

the user is referred to [Christian 82], [Goodenough 75], [Levin 77], [Yemini 85]. Our objective, instead, is to homogeneously combine into the object oriented environment described above adequate exception handling features. We begin by presenting the foundations upon which our proposed mechanism is built.

3.1 Foundations

First we aggree (as is usual in conventional languages) that the occurrence of an exception in some active unit U must cause the termination of U (unable to fulfill its contract, in the EIFELL metaphor [Meyer 88]). However, we also accept that this termination may be preceded by the execution of a last wishes section of code (a local handler), where local actions can be taken, still in the environment of the terminating unit, to flag down or cancel the exception raised. Exceptions, in this proposal, can thus be viewed also as control structures (actually labels), causing, when raised, control transfers to local handlers [Levin 77], [Cheriton 86]. In this metaphor, the concept of exception is overloaded: exceptions indicate not only abnormal situations, but also infrequent (but entirely normal) situations.
One can state, as an advantage of having local handlers for infrequent situations, the ability that programmers then have of placing "off-line" rarely used sections of code in their algorithms, consequently increasing their overall clarity. Another reason for the use of local handlers is that there may be exceptions for which, independently of the calling unit, the same treatment is acceptable - one might as well specify its handling locally.

The idea of local handling, as stated, is that found in Ada [Ichbiah 79]. We shall see shortly that the class mechanism in object oriented languages provides a natural and welcome extension to this concept.

Second, should there be no local handler, we aggree, as is the case in CLU [Liskov 79], that it is the responsibility of the calling unit to deal with the exception (dynamic scoping), as opposed to a static scoping structure, where an exception raised in U is propagated to the unit immediately enclosing U. In other words, a calling unit should be prepared to deal with both the normal and the possible abnormal behaviors of a called unit. Again the class structure modifies this idea somewhat (section 3.2).

As stated above, these foundations are supported, in full or in part, by well-known conventional languages, notably CLU and Ada. They have been chosen as the basis of our proposal also due to this familiarity - we want object oriented languages that conventional programmers can use without undue difficulties.

3.2 A Basic Architecture

Our task is to combine the foundations above with the new dimensions introduced in languages by object orientation. With respect to exception handling, we distinguish, in object orientation:

- the class structure, a static feature enclosing the definition of object behavior and created with the use of inheritance;
- the powerful semantics of object behavior, in which active units (methods, iterators, message handlers, and so forth) are applied to receiving objects.

Our reasoning begins as follows. Exceptions occur at run time, when some active unit is being executed; in object oriented terminology, when an active unit is being applied to some object. If we suppose the existence of local handlers, they are to be placed within these active units. To visualize this situation, consider the skeletal declarations below, for classes C (lines 1-6) and A (lines 7-11).

```
1     class C
      ...
2           method M
3                 A x
            begin
                  ...
4                 x <- R(...)
                  ...
5           end method
      ...
6     end class
```

```
7     class A
      ...
8           method R(...)
            begin
                  ...
9                 raise e
                  ...
10          end method
      ...
11    end class
```

In class C a method M is declared (lines 2-5). In this method a local object x of class A is declared (line 3), and method R is applied to x (line 4). This application is correct, since R is part of the behavior of x, specified in class A (lines 8-10).

Consider also that, during the execution of R on x, the exception e is explicitly raised (line 9). As stated above, what happens next depends on the existence of a local (within R) handler for e. If this handler exists, its code is executed, the exception e is flagged down, and R terminates normally. The following syntactic model could be used to express this situation, for whatever operation:

```
operation name (parameters)
      local declaration,...
begin
      statement,... (including raise)
      exception
            when exc-name,... then statement,... (including raise)
            ...
      end exception
end operation
```

In this model, the **exception...end exception** clause, a multi-way selection statement, is placed off-line, indicating that control reaches this clause only if a **raise** statement (**raise** exc-name) is executed.

Should there be no local handler for a raised exception, the raising unit should terminate without last wishes, according to the foundations described above. It is here, however, that the class mechanism in object orientation may play a significant role. We propose to use the class structure to hold yet another kind of operation: one responsible for the handling of an exception for which no local handler was defined in the raising unit.

The class could thus be seen as another scope for "local" handling - "local" in the sense that flagging down the exception would still be done before the exception is propagated to the caller, as our basic model indicates. (In the following, we shall continue to use "local handler" to indicate a handler located within the raising unit, and "class handler" to indicate the new construct being proposed.)

We illustrate by expanding class A in the example above:

```
7     class A ...
            ...
8           method R(...)
            begin
                  ...
9                 raise e
                  ...
10          end method
            ...
12          exception handler for e
                  ...
13          end handler
            ...
11    end class
```

In the example, the class exception handler for e (lines 12-13) would be automatically invoked if e is raised and no local handler for e exists. By "automatically invoked" we mean automatically applied to the object receiving R in the first place (x in the example). If we use **self** to designate the receiving object, the **raise** statement in line 9 would have the same

effect as self <- e. In fact the explicit application self <- e should be also valid as a statement, ranking with method applications, for example.

In this model, we consider the class handlers as being part of the definition of object behavior; active units applicable to self, the object which received the operation in which the exception was raised - the object for which abnormal behavior was detected.

We note at once that the semantic of method application is not an adequate substitute for the invocation of a class handler: applying a method to self, in order to treat off-line some abnormal behavior, would require the saving of a return address (that of the statement immediately following the application), to be used after the method is executed; whereas, in our basic model, the raising unit must terminate after the last wishes specified in the exception handler are executed.

Class handlers can be used to "factor out" handling common to a set of active units in a class; for example, common actions to be taken when array bounds are violated, the array being part of the data structure for the objects of the class.

It should be noted that the referencing environment in which a class handler executes does not contain the environment of the raising unit. A class handler can only reference (besides its local entities, of course) the entities available in the class itself (data to be shared among objects, for instance) and global data, if present. In this sense a local handler has more referencing power than a class handler; we are trading this power for economy of code (factoring out exception handling) and software reutilization (see remarks on inheritance below).

With respect to inheritance, class handlers behave exactly like other class units: an exception raised in some unit R for which no local handler is specified may find its handler in the class immediately containing R, or in any of its superclasses (we suppose simple inheritance in our object oriented model). An exception e is propagated to the caller (or the unit which contains the initial application of R) only if, for the entire length of the inheritance chain, no class handler is found for e. Also, the redefinition rules of the language should be applied to class handlers; and class handlers should have their definitions deferred, when convenient.

With respect to visibility, class handlers should not be exported from the class. The behavior they specify for class objects can only be invoked internally; class handlers are, by default, private.

We conclude this section on a basic model for exception handling in object oriented languages by summarizing the several handling semantics described above. We suppose the operation application

x <- R(...)

and distinguish four cases:

- no exception is raised in R's code: R terminates normally.

- an exception is raised within R, and there is a local handler for e: the handler code is executed in R's referencing environment, the exception is cancelled, R terminates and control resumes normally.

- an exception is raised within R, there is no local handler for e, but a class handler for e is found somewhere in the inheritance chain: the handler code is executed in the class' referencing environment, the exception is cancelled, R terminates and control resumes normally.

- an exception is raised within R, and there is neither a local nor a class handler for the exception: R terminates abnormally, and, for units expecting a return from R, the exception is propagated to that unit (the one containing the application statement x <- R(...)). More on propagation on section 3.3 below.

Essentially what we are proposing is a combination of static and dynamic scoping rules for the handling of exceptions. We aggree dynamic rules are preferable; however, the class structure and the accompanying inheritance mechanism provide natural grounds for static scoping as well. This is convenient since, from the point of view of classes and inheritance, class handlers require no additional semantics from that already in existence for methods, for example.

3.3 Exception propagation

In the model described above, the operation U issuing an application statement x <- R(...) in which an exception occurred may receive an exception back - an indication that, for some reason, the expected behavior was not possible for the receiving object x. This happens, as seen, when no local or class handler was found for the exception raised. In this section we discuss the handling of a propagated exception.

We begin by recognizing that the operations which are liable to receive an exception back are exactly those which expect to regain control after the application statement has been normally executed. This includes methods and iterators, for example, but it is not the case of message handlers: once a message is sent to an object, the handling of that message is done asynchronously with the execution of the unit sending the message, which does not expect to hear from the message handler - does not care whether the message was handled normally or abnormally. Clearly, for such units,

propagation is not adequate: once an exception occurs, only local or class handling is feasible.

Another initial consideration regarding the handling of propagated exceptions is their outside visibility. As opposed to locally treated exceptions, which are naturally private to a class, propagated exceptions must be visible outside the class, since they will receive outside handling. Even though compilers would not need explicit visibility declarations (every exception raised in an operation and not handled either locally or in its class is exported), for readability purposes information on exported exceptions should be provided.

One way to accomplish this is to add exception names to the **exports** list of a class:

```
class class-name
        ...
        exports field,... operation,... exception-name,...
        ...
end class
```

Another way to export an exception, more precise but perhaps less readable, would be to attach the name of the exception to the operation where it is raised. The exception could thus be considered as a part of the operation's signature (actually as a sort of qualifier to the operation, and bound to it), and the class **exports** clause could be left as before. A syntactic skeleton for this solution is offered:

```
operation name (parameters)
        exception exception-name,...
        local declaration,...
begin
        statement,...
        exception
                local handler,...
        end exception
end operation
```

To conclude this section on exception propagation, we suggest yet another level for exception handling - statement level handlers. In section

3.2 above local and class handlers were described. In both cases the actions there specified were taken immediately before the raising operation was terminated. We could add another dimension to exception handling by attaching to application statements

```
object <- operation (arguments);
```

a statement level handler, as for example in

```
object <- operation (arguments)
    exception
        when exception-name,... then statement,...
        ...
    end exception;
```

The purpose of a statement level handler would be to preserve the environment in which the application statement was issued: if the applied operation sends back an exception, and if a statement level handler exists for it, then its actions are executed, the exception is cancelled, and the statement following the enlarged application statement is executed next. This contrasts with the exception handling semantics adopted when no statement level handler is present: last wishes for the environment where the application statement was issued, should a local or class handler exist for the received exception; exception propagation otherwise.

---

4. Conclusions

We propose an exception handling mechanism for object oriented languages. We initially define the kind of language we are concerned with: one that could be considered as an extension of a conventional language; one that could be easier for real-life programmers to embrace.

We adopt well-known, conventional semantics for important issues in exception handling: the termination of the raising unit, with or without last wishes; and the dynamic propagation of exceptions not locally treated. We then recognize the important role that the class construct plays in object oriented languages. We suggest that a degree of static scoping could be naturally absorbed by this construct, and propose syntax and semantics to match. Key to the extensions proposed is the realization that abnormal behavior, expressed by exceptions and handlers, is still behavior, and hence fits elegantly in the class construct.

Bibliography

[Cheriton 86] Cheriton, David.

Making Exceptions Simplify the Rule (and Justify their Handling). **Proc.** IFIP Congress 86, Dublin, Ireland, 1986.

[Cristian 82] Cristian, Flaviu.

Exception Handling and Software Fault Tolerance. *IEEE Transactions On Computers*, 31(6) 531-540, jun 82.

[Goodenough 75] Goodenough, J. B.

Exception Handling:Issues and a Proposed Notation. *Comm. ACM*, 18(2), dec 75.

[Ichbiah 79] Ichbiah, J. et alli.

Rationale for the Design of the Ada Programming Language. *ACM Sigplan Notices*, 14(6), jun 79.

[Levin 77] Levin, Roy.

Program Structures for Exceptional Condition Handling. *PhD Thesis*, Carnegie_mellon University, jun 77.

[Liskov 79] Liskov, B., Snyder, A.

Exception Handling in CLU. *IEEE Transactions on Software Engineering*, SE-5(6) 546-558, nov 79.

[Meyer 88] Meyer, B.

Object-Oriented Software Construction. Prentice-Hall, Englewood Cliffs, NJ, 88.

[Schaffert 86] Schaffert, C. et alii

An Introduction to Trellis/Owl. Proc. OOPSLA 86, sep 86.

[Wegner 86] Wegner, P.

Classification in Object Oriented Systems. SIGPLAN Notices, v21 #10, oct 86.

[Yemini 85] Yemini, S., Berry, D.

A Modular Verifiable Exception Handling Mechanism. *ACM Transactions on Programming Languages and Systems*, 7(2) 214-243, apr 85.