Improving exception handling with Object-Oriented Programming

Christophe Dony

Equipe mixte Rank-Xerox France & LITP
University Paris VI
4 place Jussieu, 75221 Paris cedex 05.

Abstract - The aim of this paper[1] is to show and to explain how the object-oriented formalism can improve the expressive power of an exception handling system and how it can simplify its implementation and its utilization. Object-oriented design allows implementors to solve easily many classical exception handling issues such as creating user-defined exceptions, passing arguments from signalers to handlers, etc. Besides, new capabilities arise, such as defining generic protocols for handling exceptions, organizing them into a hierarchy leading to an inheritance of these protocols, signaling and handling multiples exceptions. The proposed system (implemented and used) is powerful, user-friendly, extendible and reusable without any code duplication. This paper can interest the non specialist of object-oriented programming concerned with the problem of exception handling.

I. Introduction

Exception handling techniques [3] [9] [12] [13] became an important issue because of their implications in software engineering: fault-tolerance [13] [3], modularity [13] [11] [19], 3) reusability (by extending definition domains and functionality of operations signaling exceptions), and readability (by specify in its interface all the possible responses of a module to its legal inputs).

Object oriented programming (OOP) and design (OOD), mainly initiated by *Simula* [5] and by Alan Kay works leading to *Smalltalk* [8], is a software decomposition technique that has also led to many improvements in system specification (thanks to a more natural way to specify a certain kind of problems through the description of their compound objects) and software quality (modularity, information hiding, sharing of code, reusability and extendibility) [5] [8][14].

The key-idea of the present work is to apply OOD to the specification of an exception handling system (EHS), as proposed earlier in *Zetalisp* [15], *Taxis* [17] or in [1] [16]. Our work is an extension of these systems towards a full object-oriented representation of exceptional events and of protocols designed to handle them.

The proposed concept organization first provides a promising set of solutions to most classical exceptions handling issues, such as those quoted in [10] (p. 268): ability for users to create new exceptions, to signal and handle them in the same way as system defined ones, to parameterize exceptions (i.e. to pass arguments from signalers to handlers), etc. Moreover, the expressive power of the OOP paradigm makes it possible to introduce new facilities that could not be simply implemented within procedural languages. E.g., exceptions can be first class objects and can be organized in a hierarchy that will be taken into account by the handling mechanisms; properties can be defined on exceptions; all handling operations can be performed via message sending, which makes them generic (fatal or continuable exceptions can thus be signaled with the same primitive); the hierarchy allows users to catch any set of unexpectedly raised exceptions, etc. Finally, the fundamental advantage of such a choice is that the proposed EHS inherits of the software qualities induced by the utilization of the OOP paradigm : modularity, extendibility and reusability.

The ideas described in this paper are parts of an EHS designed for OOLs [7]. However, our goal is not to discuss the exception handling requirements that are specific to OOP [7], but to present solutions that could be of interest within any EHS.

[1] This paper has been published in the proceedings of the 14th IEEE computer software and application conference (COMPSAC'90), pp. 36-42, Chicago, November 1990.

The following section will recall some basic vocabulary and concepts. After that, our system will be described in a logical order: status of exceptions, protocols for signaling and for handling. For each point, the originality of the described solution (both from the user and from the implementor point of view) will be highlighted, some examples of use and a comparison with existing systems will be provided.

II. Terminology and Concepts

A. Exception handling.

Software failures reveal either programming errors or the application of correct programs to an ill-formed set of data. An exception can be defined as a situation leading to an impossibility of finishing an operation.

The term "exception" indicates that such a situation does not always denote an error case and can be handled. The problem of handling exceptions is to provide materials allowing to establish a communication between a routine which detects an exceptional situation while performing an operation and those entities that asked for this operation (or have something to do with it). An EHS allows users to signal exceptions and to define handlers.

To *signal* an exception amounts to identify the exceptional situation, to interrupt the usual sequence, to look for a relevant handler, to invoke it and to pass it relevant information about the exceptional situation, such as the context in which the situation arose. *Handlers* are attached to (or associated with) entities for one or several exceptions (according to the language, an entity may be a program, a procedure, a statement, an expression, a data, etc). Handlers are invoked when an exception is signaled during the execution or the use of one of these protected entities.We will use the following syntax to denote the association of a handler with instructions or expressions.

```
{<protected-instructions>
  {when:<exception-names> (parameter)
      do: <handling-expressions>}}
```

To *handle* means to set the system back to a coherent state, so that standard computation can restart. Handlers have to choose, knowing about their definition context and using information provided by the *signaler*, whether to (1) transfer control to the statement following the signaling one (*resumption*), (2) discard the context between the signaling statement and the one to which the handler is attached (*termination*) or (3) signal a new exception. For further precisions on terminology and explanations on these concepts, see [9], [13] or [19].

B. Object-oriented design.

Object-oriented programs are built around *classes of objects* that implement abstract data types. Attributes of classes (usually named *slots* or *instance variables*) determine the structure of its elements. Operations related to types (usually named *methods*), are associated to classes.

Instances of classes can be though of as elements of the types; each one owns a particular value for each slot defined on its class. *Overloading* allows programmers to define methods with the same name on different domains. *Inheritance* distinguishes OOLs from modular languages like Ada. Since classes can be organized in a hierarchy, each instance of a class inherits the properties defined on its *upper classes* (also called *ancestors*)

The communication protocol (called *message sending*) takes into account (either dynamically or at compile-time) overloading and inheritance. Sending the message M to an object O with arguments a1...aN (that we will note "[O M a1 ...eN]") can be thought of as an indirect procedure call, where the procedure to be executed is the first found method, named M and the signature of which is a superset of "class(O) x class(a1) x class(aN)", arguments being checked from left to right.

One of the major contributions of OOP to software engineering is reusability. To write reusable programs with OOL can be achieved by defining polymorphic operations (valid for several data types) on abstract[2] classes (C) witch are nodes of the inheritance tree. A polymorphic operation (O) is based on lower level (concrete) operations (O1, O2), which can be defined (or overloaded when default ones have been provided) on subclasses (C1, C2) that implement concrete data types. O can then be extended to (and reused for) a

[2] A class designed to be a place where common pieces of behavior are grouped together, rather than to be instantiated.

new data type by the creation of a new subclass (C3) of C on which O1 and O2 will be redefined.

III. Improving the status of exceptions.

The first issue that arises for either the user or the implementor of an exception handling system is the status of exceptions; how are exceptions represented and referenced? How can they be manipulated or inspected? Such a status has implications on the expressive power of the whole system.

A. Exceptions and abstract data types

A transfer of control towards a label, or a specific procedure call wired into the signaler, are the most simple solutions for signaling an exception. The label or the procedure name stands for the exception; knowledge about it is embodied into the expressions associated to the label or into the procedure body. In such systems, users have no simple way to define handlers. In most of the important exception handling systems that can be found in procedural languages (e.g. *PL/I*, *Ada* [11], *Clu* [13], *Mesa* [15]), exceptions are identifiers. When an exception is raised, a handler that references this identifier is looked for and invoked.

A common characteristic of the above systems is that knowledge relative to exceptions (even the most general one) is uneasy to grasp since, in the first case, concentrated in a unique and may-be inaccessible handler, and in the second case, scattered in user-defined and default handlers provided by the system. In both cases, exceptions are not first class objects; they cannot own properties, cannot be inspected, modified, reused or enriched.

Each exception is nevertheless a complex entities of which characteristics can be given regardless of local handling considerations (for example which kind of arguments are to be passed to handlers, which message is to be reported when the exception is not handled or which interactive propositions are to be provided).

Using the class-instance model, a first, obvious and partial solution to represent exceptions consists in the creation of a class "exception" of which concrete exceptions (e.g. division-by-zero) would be some instances. This provides a place to group together behaviors common to all exceptions but does not permit to group those relevant for all occurrences of a particular exception.

A second solution which eliminates this drawback is to represent each exception as a class [16], the slots of which will represent its structure and the methods of which its functional properties. This representation also provides a solution -as does the previous one- to group together characteristics common to exceptions viewed as concepts, provided that meta-classes (classes representing and owning knowledge about classes) can be manipulated in the language (as in *Smalltalk*, *Objvlisp* [4], *Clos*, *Lore*).

B. Kernel exception classes.

Thus in our system, both exceptions viewed as concepts and occurrences of exceptions can take advantage of the object-oriented representation (cf.fig.1). Exceptions viewed as concepts are instance of the meta-class exceptionClass (a subclass of the kernel meta-class *class*) and are subclass of exceptionalEvent. Each occurrence of an exceptional situation will be modelized by an instance of a sub-class of *exceptionalEvent*, which owns their common behavior and two basic slots (cf. fig. 1& 3).

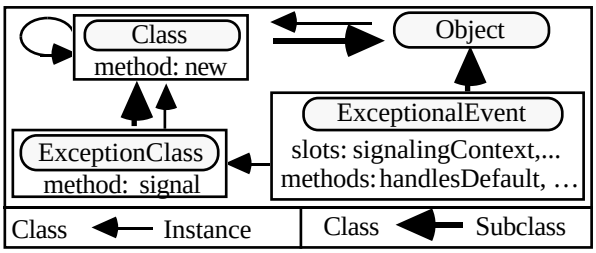

Fig.1: Kernel exception classes.

C. User-defined exceptions.

In the languages where the types "exception" is defined, users have the ability to define new exceptions by declaring identifiers to be of that type. In our system, creating a new (kind of) exception can be done by instanciating our specific meta-class *exceptionClass*.

Here is a small example where new exceptions have to be defined. Consider (1) the class *window*, with the slots *origin* and *length*; and (2) the class *terminal*, with the slot *length* and the method

displayWindow which signals an exception when a window is longer than the current terminal. Displaying a window on a screen is asked by sending to the object representing the current terminal the message *displayWindow* with the window to be displayed passed as argument. In order for the possibly raised exception to be specifically handled, a new exception class is created in the following way: "*[exceptionClass new name: windowLargerThanScreen]*".

There is no difference between system and user-defined exceptions, all can be created, raised and handled exactly in the same way. New created ones are automatically integrated into the inheritance graph. Since the main interest of such a graph lies so far in property inheritance, let us detail what does it mean to define properties on exceptions and how this can be done.

D. Properties defined on exceptions.

Two levels of knowledge about exceptional events are of interest. The former is made of all pieces of information, concerning a particular occurrence of an exception, that have to be transmitted from signaler to handlers. The latter consists in all functional properties designed to handle them.

1. Slots.

When exceptions are identifiers, there is no simple way to associate them knowledge. Whereas when exceptions are classes, a slot can be designed for each piece of information to be transmitted to handlers.

Consider, for example, our exception *windowLargerThanScreen* again, any handler for that exception is likely to make use of the characteristics of the window which was to be displayed. A slot where this information can be stored is defined on it:

```
[exceptionClass new
    name: windowLargerThanScreen
    slot windowToBeDisplayed type: window]
```

In the same way, some general purpose slots are defined on *exceptionalEvent*. For example signalingContext (cf. fig. 2 & 3) will allow each handler to know in which stack context an exception has been signaled.

2. Methods .

As stated earlier, functional knowledge is, in classical EHSs, discarded in various handlers and cannot be easily retrieved. This looks right for local knowledge, as e.g. the one embodied into a handler attached to a particular instruction (for example a handler saying that a particular call of the divide procedure should return 0 if divide-by-zero is signaled).

Besides, both the most general default handlers and the routines used within them are independent of any execution context. Instead of defining them as isolated pieces of code, our system allows to represent them as methods defined on exceptions.

Here is for example the method *handlesDefault* defined on *exceptionalEvent*, which is our system's most general default-handler, always invoked when a more specific handler cannot be found. Each selector used in this method points out a method, a default version of which being also defined on *exceptionalEvent*.

```
body of handlesDefault on exceptionalEvent
  [self[3] describeException] ; Reports the event
  [self describeContext] ; Displays its context
  [self displayPropositions] ; Displays propositions
  [self returnToTopLevel.] ;If no proposition is chosen.
```

A second example of functional knowledge defined on exception classes are those methods designed to display some interactive propositions for handling (this idea can be found e.g. in *Zetalisp*). For example, when the exception *windowLargerThanScreen* is raised, the following proposition: "*Display window's visible part.*" is displayed (cf.fig.4) by the proposition method *displayVisiblePart*. For each proposition, a joined method (in our example *doDisplayVisiblePart*) is designed to achieve actions corresponding to the proposition.

As far as global knowledge is directly attached to exception classes, inspecting them provides a good idea of what can happen when they are raised. The figure 2 shows a part of the information delivered by the inspection of *exceptionalEvent*.

[3] "self" represents the receiver of the current message., "*self describe-exception.*" means: to send the message *describe-exception* to the instance of the exception for which *handlesDefault* has been called .

```
; a slot defined on exceptionalEvent:
signalingContext  ;to be stored at signaling time
;  examples of methods defined on exceptionalEvent:
handlesDefault ;the most general default handler
describeException ;  displays a general error message
abortProposition ;  an interactive proposition
abort ; aborts the current computation.
```

Fig.2: Inspecting exceptionalEvent.

E. Exception hierarchies.

Beyond properties defined on exceptions, the key idea which underlies the choice of designing exceptions as classes are to organize them into a static hierarchy, to group common characteristics on abstract exceptions, to reuse and to extend them on concrete ones via property inheritance and overloading. For example, the exception *divide-by-zero* can be implemented as a sub-concept of the higher level exception *arithmeticException*.

Similarly, *exceptionalEvent* is divided (cf.fig.2&3) into *fatalEvent*, to which are attached generic protocols for termination, and *proceedableEvent* to which are attached those for resumption (we chose to implement a model where both resumption and termination are possible). Then, from the user's viewpoint, our system is based on three main classes: (1) *error* (cf.fig.3) is the set of exceptional events for which resumption is impossible, (2) *warning* is the set of exceptional events for which the termination is impossible, (3) finally, multiple inheritance is used to create the class *exception* for which both capabilities are allowed.

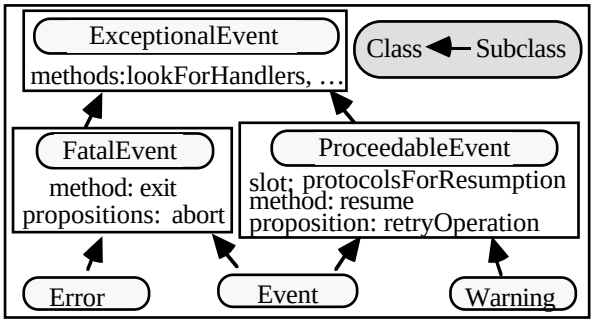

Fig.3: Top of the exception hierarchy.

Reusability and extendibility are the fundamental consequences of such an organization.

F. Inheritance.

Each new exception inherits, as soon as it is created, from the characteristics of its ancestors in the hierarchy. For example, the above displayed generic method *handlesDefault* is part of the protocol of any exception. All systems as well as user-defined exceptions can be defined as appropriate sub-classes of *error* and *warning*, depending on whether the exception is intended to be fatal or proceedable and has to own protocols for termination or resumption.

G. Reusability.

Reusability is achieved as explained in section II.2. All parts P (e.g. *describeException*) of a polymorphic method (e.g. *handlesDefault*) can be overridden at each level in the tree (say on exception X), all parts non redefined being reused. When X (or a subclass of X) is signaled, if *handlesDefault* is executed then the new method P defined on X will be executed.

We have for example redefined the method *describeException* on *windowLargerThanScreen* so that a new error message be reported to users. Considering that two new interactive propositions (No2&3) have been defined on this new exception, here is what happen when it is raised and not handled:

```
!!! W is larger than current screen. ; describeException
!!! while sending  display-window  ;  describeContext
1 : Abort.                          ; displayPropositions ...
2 : Inspect or modify W
3 : Retry the display.
4 : Display only the visible part of W.
```

Fig.4: inheriting default handlers.

This example reveals that the propositions for proceeding are also inherited. In order to highlight this point, let us complete our little window example to present a hierarchy of user-defined exceptions with embedded propositions (cf.fig.5).

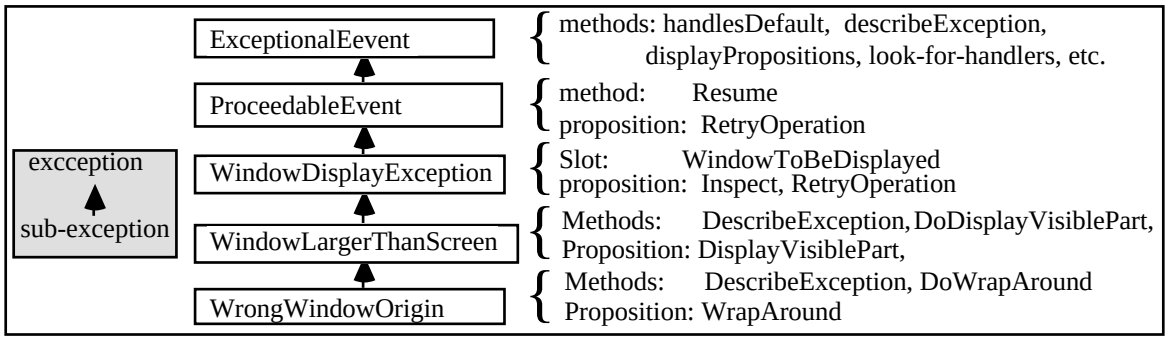

Fig.5 : A hierarchy of user-defined exceptions.

The very general *abort* proposition is defined on *fatalEvent* (not represented). The inspect and retry (P3) propositions, relevant for all exceptions raised while manipulating windows, are defined on *windowDisplayException*, a new abstract exception. The slot *windowToBeDisplayed*, itself generally relevant, is also defined on it, rather than on *windowLargerThanScreen*. Notice that P3 is a specialization of the retry proposition (P) defined on *proceedableEvent*. P3 hides P to provide a similar but more specific proposition.

A last exception in the hierarchy, named *wrongWindowOrigin*, is raised when a window comes out of the screen because its origin added to its length is higher than the screen length. When this situation is about to happen, all the previous propositions are relevant and can be inherited. Considering the screen to be circular and displaying the right-end of the window on the left of the screen is another solution for resumption. Subsequently, our last exception can be created with benefit as a subclass of *windowLargerThanScreen*. A method named *describeException* as well as the new proposition named *wrapAround* are defined on it. When *wrongWindowOrigin* is raised, the following information is reported.

```
!!! Window W : offset + length exceed screen length.
!!! While sending displayWindow to current-terminal.
1 : Abort
2 : Inspect or modify W
3 : Retry the display.
4 : Display the visible part of W.
5 : Wrap W around the screen.
```

The hierarchic organization also has consequences on the way exceptions are signaled and handled; we will develop these points in the following paragraphs.

IV. Improving protocols for signaling .

Signaling an exception in our system consists in sending it the message *signal*. Signaling is generic as far as message sending makes it impossible to signal anything but an exception. The method *signal* is defined on *exceptionClass* (cf.fig.1).

This method first calls the method *new* (cf. fig. 1) in order to create an instance of the signaled exception. The slots of this instance can be initialized at signaling time with, on the one hand values given by the signaler (e.g. *windowToBeDisplayed* or *protocolsForResumption*, cf. fig. 6), default values being used if no ones are provided, and on the other hand, values owned by the system (e.g. *signalingContext*).

Signal then sends to the created instance the message *lookForHandlers* understood by all instances of exceptions (cf.fig.3&5), which will find and invoke a handler. L*ookForHandlers* can be redefined on any new exception in order to provide, for any specific application, specific rules for searching and invoking handlers (cf. [12]). New EHSs can thus be experimented in a very simple way.

Other advantages lies in signaling simplicity, parameterisation and communication with handlers.

A. A unique signaling primitive.

Here is a piece of code in which the exception *windowLargerThanScreen* is signaled and two of its slots explicitly assigned. The slot *protocolsForResumption*, allows signalers to specify, at signaling time, which method's names, among those allowing to resume the execution, handlers will be allowed to use.

```
;  body of  "displayWindow" defined on "terminal"
….if  [[w length] >  [self length]]
   then  [windowLargerThanScreen signal
              windowToBeDisplayed: w
              protocolsForResumption: doDisplayVisiblePart]
```

Fig. 6: Signaling an exception.

```
; inspection of  an instance of  windowLargerThanScreen
type: windowLargerThanScreen
signalingContext: <an object representing a stack frame>
windowToBeDisplayed: w
protocolsForResumption:  doDisplayVisiblePart ...
```

Fig.7: Inspection of an exception instance .

It is generally agreed that a signaler is responsible for choosing between signaling either a fatal or a continuable exception whereas the handler has to say how the calculus will be restarted. Within standard EHSs, a set of primitives is generally provided to support the various signaling cases. E.g., in Goodenough's proposal, signaling with *escape* states that termination is mandatory, *notify* forces resumption and *signal* let the handler responsible for the decision.

In our system, *signal* is the unique signaling primitive. This is possible because knowing whether the signaled exception is proceedable or not only depends of its type and because all the information needed to handle it will be stored in the argument that will be transmitted to handlers.

B. Parameterization.

How to pass information from signalers to handlers is a classical issue. In standard *Ada* or *PL/I* , handlers cannot have parameters; global variables have to be used to communicate. In many other languages such as *Clu*, or *Mesa*, solutions for associating parameters with exceptions are provided; parameter names (and types) are declared either in each handler heading (*Clu*) or while declaring the exception (*Mesa*). A handler can only be invoked if the signaling arguments match the parameters types. Besides, in order to allow handlers to trap all exceptions, whatever arguments are passed, some kind of pattern-matching star conventions must be designed.

Our model provides a solution to parameterization which is both simple to use and to implement. All the handlers receive a unique argument which is the previously described instance of the current exception. This argument is a structured object holding all pieces of information about the exception. Here are the pieces of information about the signaling time created instance of the above signaled exception that can be accessed within a handler.

C. Cooperation for resumption.

Resumption raises another issue: it needs a communication between the signaler and the handler. Although the handler is responsible for saying what to do, it may happen that the operations allowing to restart the computation must be performed by the signaler in its environment. In such a case, the signaler might want to predict which kind of resumption have been implemented. Not all systems allowing resumption have taken such issues into account; *PL/I*, *Mesa* or Goodenough's proposal did not, execution simply restart at the expression following the signaling one. [19] allows users to states, in procedure headings, the type of results that may be returned by handlers. Iso *Lisp* proposal [18] provides powerful control structures allowing signalers to write named pieces of programs to be executed in the signaling environment after resumption, these names can be used within handlers to entail resumption.

Our solution is to define on *proceedableEvent* a slot named protocolsForResumption, allowing signalers to specify, at signaling time, which method's names, among those allowing to resume from the current exception and needing the collaboration of the signaler, can be used in this particular case (cf. fig. 6 & 7).

D. Signaling multiple exceptions.

Signaling multiple exceptions is useful either to avoid the creation of a specific class or to report very general situations. Within our system, signaling multiple exceptions can be done by sending the message *signal* to exceptions that are nodes in the class hierarchy. Here is an example where a handler achieves information hiding by trapping a low level exception and by re-signaling a semantically higher level one.

```
{body of a method
  {when: overflow (e) do: [arithmeticException signal]}}
```

Let us now see, while describing protocols for handling, how multiple exceptions can be trapped too.

V. Improving protocols for handling

Handlers are responsible for saying what to do after the occurrence of an exceptional situation. Our default handlers are defined on exceptions classes, they are grouped together, are inherited and can be reused (cf. §III.G). Other handlers are associated to expressions. All handlers have a unique parameter bound to the instance of the current exception and through which signaling arguments are conveyed. Let us now see how to handle multiple exceptions and to write handlers in a generic way.

A. Handling multiple exceptions.

All handlers are aware of the exception hierarchy, defining a handler for an exception amounts to defining a handler for all exceptions that are subclasses of it. In order to determine whether a handler is to be invoked, the system simply tests whether the exception it references is an upper type of the signaled exception. Thus, any (may be unexpected) exception which is a subclass of the exception for which the handler has been designed, can be caught by defining a sole handler. For example, any handlers for *windowDisplayException* catches just as well *windowLargerThanScreen* or *wrongWindowOrigin*.
Notice that the current exception can be signaled again, even though being handled in a multiple way, by writing: "*when: windowDisplayException (e) do:[[e type] signal]*"

B. Handling in a generic way.

In Goodenough's proposal, programmers cannot define handlers without knowing whether they will trap a fatal or a proceedable exception.

In *Mesa*, various signaling primitive are provided but handlers can be defined independently of how the exception is signaled. The system then has to check that the actions undertaken within handlers are compatible with the signaling primitive that has been used.

In our system, the set of actions that can be performed by handlers only depends on the signaled exception. They depend on the one hand of its position in the hierarchy and on the other hand of the content of its slots (e.g. *protocolsForResumption*). Any method allowing users to put the system back into a coherent state have to be invoked within handlers via message sending to the handler argument. Inheritance and message sending rules make these invocations generic.

Genericity first means that implementors do not have to perform tests to ensure that operations not compatible with the signaled exception will not be invoked. Resumption (resp. termination) is achieved by sending the message *resume*„ (resp. *exit*). As the corresponding method *resume* is defined on *proceedableEvent*, sending the message to an object which is not an element of *proceedableEvent* will automatically fail.

Genericity also means that operations relevant to the current exception will automatically be selected for each handling action. When the message *handlesDefault* is sent to the handler argument, the most specific method (according to *the argument* class) of that name will be invoked, even though the handler does not know which exception has been trapped (multiple exceptions). Thus if *windowLargerThanScreen* is trapped by a handler for *windowDisplayException*, all methods specific to the trapped exception will anyway be invoked.

VI. Conclusion

In this paper, we have explained how giving exceptions the status of abstract data types hierarchically organized and to occurrences of exceptional events the status of instances of those types, fits well to implement a user-friendly, powerful, extendible and reusable exception handling system. We have described a specification of such a system using an OOL. Several classical exception handling requirements have been easily implemented and new functionalities have been proposed, among which the most important is perhaps the ability for users to reuse and customize the existing system in order to provide specific exception handling in application programs.

The system has been implemented in the *Lore* [2] object-oriented language and in *Smalltalk*, the *Smalltalk* version is public-domain. See [6] for a detailed description of the implementation. Here is an example of how to associate a handler to an instruction in our Smalltalk implementation:

```
[currentTerminal displayWindow: w]
    when: wrongWindowOrigin
    do: [:e | e wrapAround]
```

Acknowledgement

I would like to thank Michel Bidoit for its comments and suggestions.

References

[1] A.Borgida : Exceptions in Object-Oriented Languages. ACM Sigplan Notices, Vol. 21, No. 10, pp. 107-119, October 1986.

[2] Y.Caseau, C.Benoit,, C.Pherivong: Knowledge Representation and Communication Mechanism in Lore. Proc. of ECAI'86, Brighton, July 1986.

[3] F.Christian : Exception Handling and Software Fault-t Tolerance, IEEE Trans. on Computers, Vol. C-31, No. 6, pp. 531-540, June 1982.

[4] P.Cointe: Metaclasses are first classes, the Objvlisp model. Procs. of OOPSLA'87, Orlando, Sigplan Notices, Vol. 22, No 12, pp. 156-167, October 1987.

[5] O.Dahl, B.Myhrhaug, K.Nygaard: SIMULA-67 Common Base Language. SIMULA Information, S-22 Norwegian Computing Center, Oslo, Norway, October 1970.

[6] C.Dony: Langages à objets et génie logiciel, applications à la gestion des exceptions et à l'environnement de mise au point. Thèse de l'université Paris VI, Mars 1989.

[7] C.Dony: Exception handling & Object-Oriented Programming: Towards a Synthesis. Joint conference Ecoop-OOPSLA'90. Ottawa, Oct. 1990.

[8] A. Goldberg, D. Robson: SMALLTALK 80, the language and its implementation. Addison Wesley 1983.

[9] J.B.Goodenough : Exception Handling: Issues and a Proposed Notation. Communication of the ACM, Vol. 18, No. 12, pp. 683-696, December 1975.

[10] E. Horowitz: Fundamentals of Programming Languages. Springer Verlag, Berlin-Heidelberg, New York, 1984.

[11] J.Ichbiah & al : Preliminary ADA Reference Manual. Rationale for the Design of the ADA Programming Language. Sigplan Notices Vol. 14, No. 6, June 1979.

[12] R.Levin : Program structures for exceptional condition handling. Ph.D. dissertation, Dept. Comp. Sci., Carnegie-Mellon University Pittsburg, June 1977.

[13] B.Liskov, A.Snyder : Exception Handling in CLU. IEEE Trans. on Software Engineering, Vol. SE-5, No. 6, pp. 546-558, Nov 1979.

[14] B.Meyer: Object-oriented software construction. Prentice-Hall, 1988.

[15] J.G.Mitchell, W.Maybury, R.Sweet: MESA Language Manual. Xerox Research Center, Palo Alto, California, Mars 1977.

[16] D. Moon, D. Weinreb : Signalling and Handling Conditions, LISP Machine Manual, MIT AI Lab., Cambridge, Massachussets, 1983.

[17] B.A.Nixon : A Taxis Compiler. Tech. Report 33, Comp. Sci. Dept., Univ. of Toronto, April 83.

[18] K.Pitman: Error/Condition Handling. Contribution to WG16. Revision 18.Propositions pour ISO-LISP. AFNOR, ISO/IEC JTC1/SC 22/WG 16N15, April 1988.

[19] S.Yemini, D.M.Berry : A Modular Verifiable Exception-Handling Mechanism. ACM Trans. on Progr. Languages and Systems, Vol. 7, No. 2, pp. 213-243, April 1985