Exception Handling and Object-Oriented Programming: towards a synthesis.

Christophe Dony

Rank-Xerox France & LITP

University Pierre & Marie Curie (Paris VI)
4 place Jussieu, 75221 Paris cedex 05.
chd@rxf.ibp.fr

Abstract

The paper presents a discussion and a specification of an exception handling system dedicated to object-oriented programming. We show how a full object-oriented representation of exceptions and of protocols to handle them, using meta-classes, makes the system powerful as well as extendible and solves many classical exception handling issues. We explain the interest for object-oriented programming of handlers attached to classes and to expressions. We propose an original algorithm for propagating exceptions along the invocation chain which takes into account, at each stack level, both kind of handlers. Any class can control which exceptions will be propagated out of its methods; any method can provide context-dependant answers to exceptional events. The whole specification and some keys of our *Smalltalk* implementation are presented in the paper.

Keywords: exception handling, fault-tolerance, class-handlers, expression-handlers, exception propagation algorithm, modularity, reusability, knowledge representation, object-oriented design, interactive debugging.

1. Introduction

Modern languages dedicated to software engineering usually provide exception handling systems (EHS). Different strategies for exception handling can be found in today's object-oriented languages (OOL) but no standard solution exists. This paper presents a specification of an EHS, designed with and for OOLs, that attempts to synthesize the main advantages of the existing systems. It is based on the three following remarks.

1) The work carried out for handling exceptions in procedural languages such as *PL/I* [PL/I 78], *Clu* [Liskov79], *Ada* [Ichbiah79] or *Mesa* [Mitchell79], has demonstrated that modularity and the notion of fault-tolerant encapsulation rely on a stack-oriented research of handlers, on the ability for signalers to raise exceptions to the operation callers, and for the callers to handle the exceptions raised by inner modules.
2) *Smalltalk-80* [Goldberg&al83] has shown the utility of attaching handlers to object classes (that we will call class-handlers), providing an object-oriented style of handling exceptional situations and of reusing code in which exceptions are signaled.
3) [Nixon83], [Moon&al83], [Borgida86] [Pitman88] have proposed systems, where exceptions are hierarchically organized classes, that significantly improve classical EHS.

Our goals are to synthesize, extend and simplify the above quoted systems in order to provide a user-friendly, powerful, extendible and object-oriented EHS. We will discuss the following issues: (1) What are the advantages of an object-oriented representation of exceptions. (2) How to integrate signaling and handling protocols into an exception hierarchy? (3) How to achieve modularity in presence of exceptions and (4) how to specify responses to exceptions at the class and at the method level?

Section 2 introduces our terminology. Section 3 deals with the status of exception. We recall how some well-known exception handling issues [Horowitz 84 p. 268] are easily and efficiently solved by representing exceptions as hierarchically organized classes. Our kernel exception classes with their associated protocols and an example of a hierarchy of exceptions are presented. Section 4 presents how to signal exceptions with a unique signaling primitive and how to parameterize them. Section 5 explains how to write handler bodies in a generic way. Why both handlers associated with classes and with expressions should be proposed within an object-oriented EHS, why they all should have a dynamic scope and how to implement them are issues discussed in section 6. Section 7 proposes some examples of how to define our various kind of handlers. Finally, related works are discussed.

The proposed specification is relevant for any OOL with some restrictions: (1) it takes advantage of the notion of meta-class, a simplified version can be implemented provided that class methods can be defined in the language; (2) some issues related to default handling only make sense within an interactive programing environment.

Two versions of the system have been implemented: the former [Dony 88] [Dony 89] for the *Lore* language [Caseau86] [Caseau87], the latter for *Smalltalk*[1] (some notes about this second implementation are provided throughout the paper). All examples in the paper use the *Smalltalk* syntax.

---

[1] *Objectworks for Smalltalk-80, v2.4, v2.5* [PP 88].

2. Terminology

Software failures reveal either programming errors or the application of correct programs to an ill-formed set of data; more generally, an exception can be defined as a situation leading to an impossibility of finishing a computation. Two main types of exceptions can be distinguished [Goodenough 75] : the domain exceptions raised when the input assertions of an operation are not verified and the range exceptions raised when the output assertions of an operation are not verified or will never be.

An EHS provides materials and protocols allowing programmers to establish a communication between a routine which detects an exceptional situation while performing an operation (a signaler) 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 (1) identify the exceptional situation, (2) to interrupt the usual sequence, (3) to look for a relevant handler and (4) to invoke it while passing it relevant information. Handlers are defined on (or attached to, or associated with) entities for one or several exceptions (according to the language, an entity may be a program, a process, a procedure, a statement, an expression, etc). Handlers are invoked when an exception is signaled during the execution or the use of a protected entity.To handle means to set the system back to a coherent state i.e. either (1) to transfer control to the statement following the signaling one (resumption) or (2) to discard the context between the signaling statement and the one to which the handler is attached (termination) or (3) to signal a new exception.

For further precisions and explanations, see [Goodenough75], [Liskov&al79] [Yemini&al85], [Knudsen87] or [Dony89].

3. Status of exceptions.

3.1. Exceptions as classes.

The first issue that arises for either the user or the implementor of an EHS is the status of exceptions. How are exceptions represented and referenced? How can they be manipulated or inspected? Exceptions are usually strings, symbols or variable of type "exception" (as e.g. in *Ada* [Ichbiah&al79]) that cannot be inspected or enriched. General knowledge relative to exceptions (e.g. default-handlers) is uneasy to grasp since unaccessible or scattered in various handlers.

Exceptions are nevertheless complex entities of which descriptions can be given regardless of local handling considerations. A first solution for representing exceptions within an OOL is to create 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 offer any opportunity to particularize the behavior of each of them. A second solution which eliminates this drawback is to represent each exception as a class (cf. *Taxis* [Nixon83], *Zetalisp* [Moon&al83] or [Borgida86]); grouping together the characteristics common to exceptions viewed as concepts is again possible provided that meta-classes exist in the language

In our system, each occurrence of an exception is an instance of a subclass of the class *ExceptionalEvent*, which owns their common behavior and determines their basic structure.

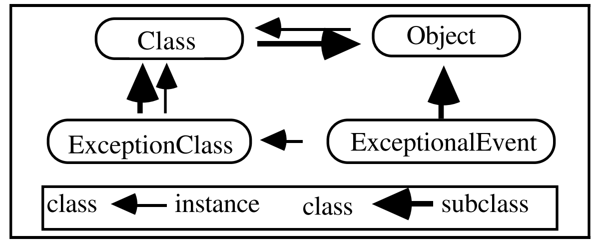


Fig.1: Kernel exception classes.

All exceptions viewed as concepts are instances of the meta-class *ExceptionClass*[2] and are subclass of *ExceptionalEvent.* Both exceptions viewed as concepts andtheir occurrences are then first class objects, can be inspected, modified or enriched (cf.fig.1).

This representation entails numerous advantages:

- New exceptions can be created as subclasses of *ExceptionalEvent*. There is no distinction between system and user-defined ones, all can be signaled and handled in the same way.
- Pieces of information concerning an occurrence of an exception can be stored on slots defined on the related class (e.g. all components of a message the execution of which has failed, are stored on the slots defined on the exception *IncorrectMessage* (cf. fig.2).
- Pieces of information designed to handle exceptions, that are independent of any execution context, can be defined as methods on exception classes. Examples are default handlers (cf.§.7.3), or methods providing standard solutions for handling. For example (cf.fig.2) the method *newReceiver:* and the associated proposition (cf.§.7.3) *askForNewReceiver* provide users with general pre-defined protocols to resume after occurrences of *WrongMessageReceiver.*
- Beyond properties defined on exceptions, the key idea which underlies the choice of designing exceptions as classes is to organize them into a hierarchy which makes the system extendible and reusable. For example (cf.fig.2), when created as a subclassof *WrongMessageReceiver* and *WrongMessageSelector*, the exception *DoesNotUnderstand* inherits three pre-defined protocols for handling.

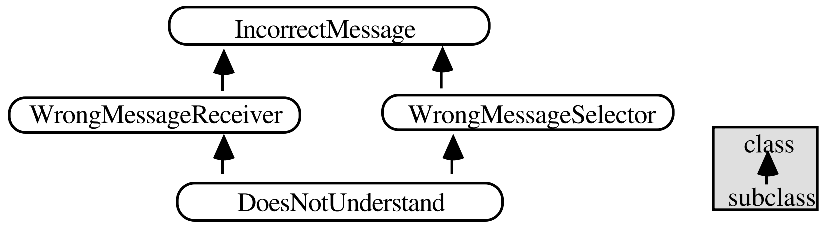
As we will see now, other advantages of this class-oriented organization lie in the way exceptions can be signaled and handled.

3.2. Kernel exceptions and associated protocols.

All basic protocols to handle exceptions (*exit*, *resume*, *retry*, and *signal*) are implemented by methods defined on kernel exception classes (cf.fig.3). *ExceptionalEvent* is then divided into *FatalEvent*, to which are attached propositions and methods for termination, and *ProceedableEvent* to which are attached those for resumption.

---

[2] When explicitly manipulated, meta-classes are *Class* subclasses [Cointe87]. In our *Smalltalk-80* implementation, *ExceptionClass* is implemented by the automatically created meta-class *ExceptionalEvent class*. Each exception class has its own (automatically created) meta-class that will inherit of *ExceptionalEvent class*.




```
Event subclass: #IncorrectMessage              "cf. fig. 3 for visualizing the exception- class Event"
    instanceVariableNames: 'messageReceiver messageSelector messageArgs '
methodsFor: 'proceeding'
    askForMessageValue...                      "ask for a value to be used as the message result ..."
    messageValue: ...                          "resume with the message value ..."

"An exception to be signaled when the selector is obviously wrong."
IncorrectMessage subclass: #WrongMessageSelector ...
methodsFor: 'proceeding'
    askForNewSelector ...                      "ask for a new selector ..."
    newSelector: ...                           "resume with the new selector ..."

"An exception to be signaled when the receiver is obviously wrong."
IncorrectMessage subclass: #WrongMessageReceiver ...
methodsFor: 'proceeding'!
    askForNewReceiver ...                      "ask for a new receiver ..."
    newReceiver: ..."                          "resume with the new receiver ..."

"a class with multiple inheritance  to be signaled when both alternatives are possible"
Class named: #DoesNotUnderstand ...
    superclasses: 'WrongMessageReceiver WrongMessageSelector '
     ...
```

Fig.2: An exemple of a hierarchy of exceptions

From the user's viewpoint, the system is then based on three main classes:

- *Error* is the class of exceptional events for which resumption is impossible;
- *Warning* is the class of exceptional events for which the termination is impossible;
- finally, multiple inheritance is used to create the exception-class *Event* in order to allow both capabilities.

The slot *signalingContext* will be dynamically bound at each occurrence of an exception to the signaling context. The slot *propositions* (instance variable of the meta-class) is used to store for each exception some propositions for interactive handling (cf.§.7.3).

4. Signaling.

Signaling any exception consists in sending to the related class the message *signal* defined on *ExceptionClass* (cf.fig.3), e.g.:

DoesNotUnderstand signal.

Passing arguments. One argument can be passed for each slot defined on the exception, e.g. after an attempt to perform the message "1 plusse: 2", the system would signal the exception *DoesNotUnderstand* in the following way:[3]

```
DoesNotUnderstand
    signalWihMessageReceiver: 1
            messageSelector: #plusse
            messageArgs: #(2).
```

- The method *signal* first calls *new* in order to create an instance of the signaled exception - that will be called the "exception object".
- *Signal* then sends to the exception object various initialization messages to assign its slots with, on the one hand values given by the signaler (e.g. *messageReceiver*), and on the other hand, values owned by the system (e.g. *signalingContext*).

---

[3]The idea is to allow users to provide a list of alternating initialization arguments names and values. This made no problem in our *Lore* implementation since the method *new* was able to receive a variable number of arguments. As *Smalltalk-80* does not allow methods to have a variable number of arguments, users have to define on each exception class a method named "*signalWithIv1Name:Iv2Name:...IvnName:*"; however, they can be automatically generated thanks to meta-classes [Goldberg&al 83] [Cointe 87].

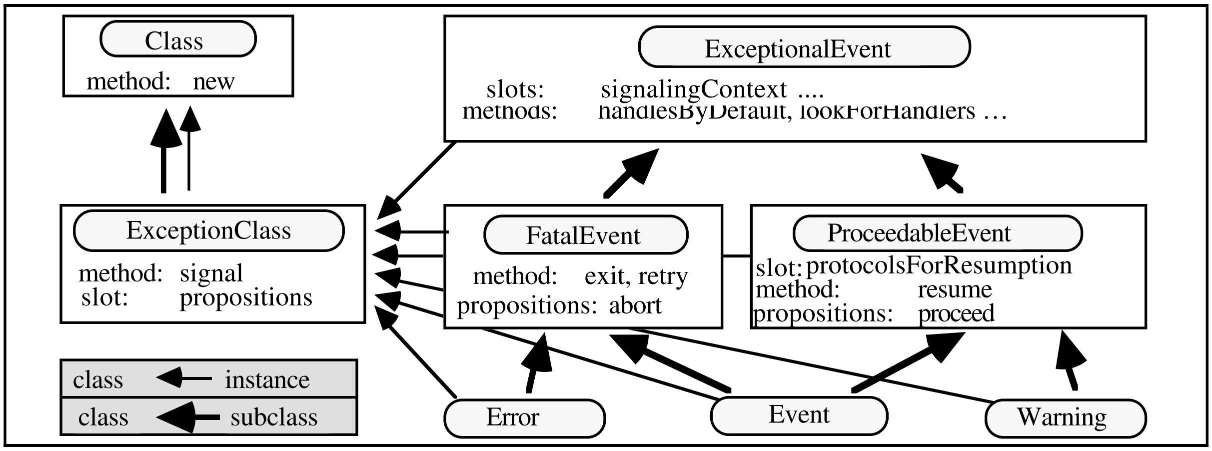


Fig.3: Kernel exception classes, associated slots and methods.

- *Signal* finally sends to the created and initialized instance the message *lookForHandlers* understood by all instances of exceptions (cf.fig.3), which will find and invoke a handler[4].

Protocols for signaling. 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 single basic 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.

Cooperation for resumption. Resumption raises a specific issue, it should not be achieved without the agreement of both the signaler and the handler when, although the handler is responsible for saying what to do, the operations allowing to restart computation must be performed by the signaler in its environment. In such cases, the signaler might want to predict which kind of resumption are possible. A slot named *protocolForResumption*, defined on *ProceedableEvent* provides a basic solution to this problem. The signaler can use it to indicate, at signaling time, the options among which a handler may choose to achieve resumption. Assigning it to *nil* means that resumption is impossible[5].

```
result <-
  DoesNotUnderstand
    signalWithProtocolsForResumption:
                 #(supplyValue newReceiver newSelector)
                 messageReceiver: anObject
                 messageSelector: aSymbol
                 messageArgs: anArray.
; result is an association <option, value>
; if control returns there, the variable result will be tested
;  and appropriate actions will be performed
```

This slot will be accessible within handlers and checked by the system before effective resumption.

[Pitman 88] has proposed for to this problem a more sophisticated solution: some new dedicated control structures (e.g. *restart-case*) provide a user-friendly way (with a case-like syntax) of writing code such as the one in the example and allow users to dynamically create new options for proceeding.

5. Writing handlers bodies in a generic way.

Handlers are responsible for saying what to do after the occurrence of an exceptional situation. All kind of handlers (cf.§.7) can use in the same way the same protocols to put the program execution back into a coherent state. Important ideas, allowing to improve classical ways of handling, come again from *Zetalisp*:

- All handlers have a unique parameter, automatically bound at handling time to the instance of the current exception and through which arguments provided by the signaler are conveyed[6]. This solves one of the main problem of classical EHS.
- Multiple (abstract classes) exceptions can be caught by defining a sole handler. Any (may be unexpected) exception which is a sub-exception of the exception for which a handler has been designed will be trapped.

We have added the idea that handling should only be performed via message sending to the exception object, all protocols for (default) handling being defined on exception classes (cf.fig.3) and inherited. Four basic ways of handling an exception are provided: resumption, termination, signaling a new exception or propagating the trapped one (cf. § 7 for examples). When handlers do not choose explicitly one of these solutions, the exception *ExceptionNotHandled* is signaled.

- **Termination**: sending to the exception object the message "*exitWith: <aValue>*" entails termination. The execution contexts between the signaler and the handler are discarded

[4]From a structural point of view, a parallel can be drawn between the couple "*signal - lookForHandlers*" and the classical couple "*new - initialize*" for creating and initializing objects.

[5] The default value of the slot is *#(resume)*.

[6] Accessing these arguments supposes that slot accessors are defined on each exception; these accessors can be automatically generated.

while recovery blocks are executed[7]. The argument's value becomes the value returned by the expression to which the handler was attached. Class handlers are invoked when an exception is about to be propagated outside of a method defined on a protected class; entailing termination within a class handler causes this method to return the exit value. Default-handlers, defined on exception classes, are conceptually attached to the top-level loop, thus exiting from a default-handler returns to the current top-level.

Retry [Mitchell 79] is a variation of termination, the protected expression is re-executed under the same protections after contexts has been discarded.

- **Resumption**: sending to the exception object the message "*resumeWith: <aResumptionOption> with: <aValue>*" entails resumption. The couple <option, value> becomes the value returned by the method *signal* provided that the option belongs to the *protocolsForResumption* collection.
- **Explicit propagation**: signaling a new exception within a handler is a crucial possibility for modularity (cf.§.6.2). This can be done by sending the message *signal* either to a new exception (a class), or to the exception object in order to propagate it to outer handlers[8]. In both cases the handler research starts from the context in which the current handler has been invoked.
- Entering the debugger can be done by sending the message *handlesDefaut* to the exception object (provided that the method has not been redefined), however this is not a basic way to handle since the debugger will further abord or resume the computation.

Thanks to message sending, handling is a generic operation. Genericity first means that neither programmers nor implementors have to perform tests to ensure that operations incompatible with the signaled exception will not be invoked[9]. For example, any attempt to send the message *exitWith:* to an object which is not an element of *FatalEvent* will fail. Genericity also means that the operations relevant to the current exception will automatically be selected even though an abstract (a multiple) exception has been caught.

[7] An important issue when both resumption and termination are possible is the restoration of valid contexts. Indeed, signalers as well as handlers that propagate exceptions do not know whether there will be resumption or not. Thus they do not know whether they have to perform some restorations (cf. cleanup handlers [Goodenough 75]). Our system provide a lisp *unwind-protect* like primitive, allowing users to specify restoration actions to be executed only when the stack frame in which they are defined is really discarded. While looking for a handler (a non destructive activity in our system), we simply mark all stack frames containing recoveruy actions. When no automatic garbage collection exists, it should also be possible to associate such restoration actions with classes (see e.g C++ destructors).

[8] This appears to be slightly different than signaling the exception again, no new object being created and all pieces of information about the original event being left unmodified.

[9] This rule is violated for resumption where the slot *protocolForResumption* is tested by the system.

6. Which kind of handlers and how to look for them.

In most languages, handlers are associated with code (instructions in *Clu*, method's bodies in *ADA*, expressions in *Zetalisp*, etc) and have a dynamic (restricted to one stack level in *Clu*) scope[10]. The main interest of dynamic handlers is modularity: exceptions are propagated to the operation callers which are thus able to specify context dependant answers or to hide the details of their implementation. The purpose of this paragraph is to highlight the interest of another kind of handlers which are associated with object classes, to explain why they fit the object-oriented design style and what should be avoided to make them really efficient.

6.1. Interest of class-handlers.

Smalltalk-80 has promoted an original class-oriented vision of handling. Exceptions are signaled within methods by sending to the current receiver, a message (e.g. *error:*) corresponding to the current exception [Goldberg&al 83 p.102]. *Smalltalk* class-handlers are standard methods, they have a static scope because determining which handler will be invoked after a signal can be done statically by inspecting the class (and its ancestors) in which the method which signals the exception is defined. Let us see why class-handlers fit the object-oriented design.

- Class-handlers allow all objects of a particular class to react in the same way when applied methods encounter an exceptional situation. They allow users to specify, at the class level, which exceptions can be propagated outside of methods defined on that class, and on its subclasses. For example, they allow an implementor to state that *overflow* and *emptyStack* are the only exceptions that should be propagated out of a method defined on the class *Stack*. Any other exception being either the result of a misuse or of a bug in the implementation, in such cases the implementor might want the program to be stopped or the debugger to be entered.
- Since class-handlers are class properties, classical OOP reusability schemes can be applied. Consider again the class *Stack*. Now suppose that we want to implement a class of stacks that are able to grow[11] when needed. A solution to this problem is to create a *Stack* subclass named *GrowingStack*, on which will be defined a handler for *overflow* and a method *grow*, this handler can resume the interrupted method, whatever its name and its location, after having grown the stack (cf.§.7.2).

  Similarly, class-handlers allow a new subclass to handle a particular exception regardless of the number of methods, upper in the hierarchy, in which it is raised. Suppose we want to create, in *Smalltalk*, a class of ordered collections for which particular actions can be performed when items are searched for and not found in the collection, either while accessing to, searching or removing elements: all cases where the exception *ItemNotFound* is signaled. To define a single class-handler for *ItemNotFound* on a subclass of *OrderedCollection* fulfils the requirements.
- It is sometimes argued that class handlers are only useful to trap the exception *doesNotUnderstand* (to perform actions

[10] Dynamic extent and indefinite scope.

[11] This is appropriate when the default size meets the current applications needs.

around message sending [Pascoe 86] (cf. § 7.2). The main reason is certainly that, in *Smalltalk*, all other exceptions are raised with the sole selector *error:*. Although some domain tests are performed via message sending, we think that, within OOLs, all exceptions should be explicitly named since many of them are not converted into the exception *doesNotUnderstand*. Consider e.g. : 1. the range exceptions, 2. those raised when the input assertions of an operation are not only based on argument types, 3. those raised when argument types are not classes of the system (e.g. positive integer). Assuming that all exceptions have a name, it would become natural to define, as shown before, class-handlers for them.

6.2. Weaknesses of class-handlers.

However, class-handlers, as they are invoked in *Smalltalk*, have an important drawback: their static scope which entail problems with modularity. As far as exceptions are not propagated to method callers, a method has no way to regain control, either to hide the occurrence of an exception (modularity) or to execute some recovery actions, when one of the methods it has invoked failed.

Consider the following method M1 defined on class C1 (e.g. the class *View*) which uses composition (one of its slot is a compound object to which messages are sent, e.g. with the MVC, view's methods send messages to the model) and suppose that M2, defined on class C2 (e.g. class *Model*), signals an exception by sending the message *error:* to *self*. Although a handler (method *error:*) is defined on C1, it will never be invoked. M1 has no way to trap the exception signaled within M2. It can neither handle the situation nor hide its implementation details to its clients (e.g the fact that it send messages to its model).

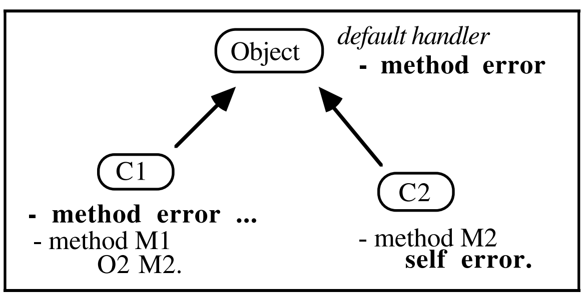


Fig.4 : composition and modularity

Two conditions are necessary for a class-handler on C1 to be invoked: 1. exceptions have to be propagated along the invocation chain, 2. default handlers (method *error:* on *Object*) should not be defined on top of the classes hierarchy.

Besides, class-handlers have another drawback, they cannot give context dependant answers. Even if the method error on C1 had been invoker, it could not access M1 local variables nor specify context-dependant answers to exceptions. This proves that, although class-handlers are useful, they cannot replace handlers associated with expressions and knowing about execution contexts.

Smalltalk programmers were used to circumvent all class-handlers problems by defining methods accepting a block to be executed in exception cases (e.g. *at:ifAbsent:, compile:ifFail:,* etc.). The new *Objectworks Smalltalk* EHS [PP 89] solves the second problem but not the first one (cf. § 8).

6.3. Synthesis: how to look for handlers.

We have seen that various kind of handlers can be of interest within an OOL provided that appropriate rules for searching them are designed. We propose a system where both handlers associated with expressions (created into the stack at expression execution time) and with classes (stored on classes at handler definition time) can live together, all having a dynamic scope. What determines the scope of handlers is the way they are searched. The method *lookForHandler* (cf.fig.3) propagates exceptions along the invocation chain by performing the following algorithm.

> Let C be the stack context in which *signal* has been performed and E be the signaled exception. If C has been established by an invocation of the method *when:do:*[12] and if the associated handler H traps E, then execute H. Else, let CL be the class of the receiver of the current context method, if a handler H for E is defined on CL, then execute H. Else go to the next frame[13] (i.e. C's sender) and loop.

This algorithm allows all the above quoted applications of class-handlers to work and eliminates their drawback as far as (1) our default handlers are defined on exception classes (cf. § 7.3) (in the previous example, a class handler on C1would be invoked) and (2) expression handlers can be defined when context dependent answers are needed. It allows classes to trap exceptions propagated out of their methods but gives a priority to handlers attached to expressions within these methods. It has been implemented in *Smalltalk-80 v2.4* thanks to the dynamic pseudo variable *thisContext* which provides at any time a pointer towards an object representing the current context.

6.4. How to store handlers on classes.

We have implemented class-handlers as some kind of methods (instances of the class *ClassHandler* which is a subclass of *CompiledMethod*). Instances of *ClassHandlers* are not associated with selectors and should not be called by message sending but only by the system after an exception has been raised. Besides, while looking for handlers, we have to determine very rapidly whether a class is protected against a particular exception. For these two reasons, we do not store class handlers in the standard method dictionaries but in a slot named *classHandlers* defined on *ClassDescription* and owned by each class. In order to achieve inheritance of class-handlers and to increase the access speed, their inheritance is statically computed, i.e. a class handler created on C is propagated and copied at define time on all C subclasses[14]. The idea is to get, with one slot access, an ordered collection of all the class-handlers defined on and inherited by a class. This solution was easy to achieve in our first *Lore* implementation and we have ported it in *Smalltalk*. Another approach would be to use caches, we do not have yet investigate in that direction.

---

[12] allowing to associate handlers with expressions (cf.§.7.1)

[13] The real algorithm is a little more complex since we have to ensure that when exception are signaled within handlers (either expression or class ones), handlers are not recursively invoked.

[14] This technique has been used to speed up the method look-up, a detailed description can be found in [Caseau 87].

7. Protocols for defining handlers.

We describe here how to define the three kind of handlers provided in the system.

7.1. Expression handlers.

Expressions handlers can be attached to any kind of *Smalltalk* expressions by using the method *when:do:* defined on *BlockClosure*, with the following syntax:

```
[<protected expression>]
   when: <exception-name>
   do: [<handler parameter> | <handler body>]
```

The first argument is the exception to be trapped and the second one is the handler block. Since blocks are implemented as lexical closures, handlers are automatically executed in the lexical environment in which they have been defined. The method *when:do:* creates a new context in the stack (that will be used to search handlers) and executes its receiver. Defining an expression handler costs two lexical closures and one method invocation.

Here is a first example showing a way to use a *stack* able to increase its allocated space. It highlights the ability to access the environment in which the handler is defined.

```
[aStack push: anInteger]
   when: Overflow
   do: [:anOverflow |  aStack grow. anOverflow retry].
```

The following example highlights the ability to trap a low level exception and to propagate a higher level one. It expresses that a process may only be suspended if it belongs to the collection of active processes.

```
[ActiveProcessList remove: aProcess]
   when: ItemNotFound
    do: [:anItemNotFound | ProcessWasInactive signal]
```

Here is a final example using *exit* and *retry* where two handlers are attached to the same expression using the method *when:do:when:do:*. Up to five handlers can be attached to the same expression with the same semantic than in *Clu* or *Ada*. The idea is here to create an evaluation loop protected against all exceptions in order to design either a top-level or a workspace. The only way to exit the loop is to signal the exception *LoopExit*.

```
[[true] whileTrue: [self body]]
    when: LoopExit
    do: [aLoopExit: | aLoopExit exitWith: #bye]
    when: ExceptionalEvent
    do: [anExcEvent: | anExcEvent handlesByDefault.
                       anExcEvent retry]
```

7.2. Class handlers.

Class handlers can be attached to any *Smalltalk* class by using the method *when:do:*, defined on *ClassDescription*, with the following syntax:

```
<protected class>
   when: <exception-name>
   do: '<handler parameter> | <handler body>'
```

The first argument is the exception to be trapped and the second one is the handler string. This method *when:do:* first calls the smalltalk compiler to compile the handler string in the environment of the protected class so that instance and class variables defined on the class can be used within the handler. Then it inserts the created handler in the handler collection of the class and of its subclasses. For each class, class-handlers are ordered compared to the exceptions they are defined for. Class-handlers cost nothing while exceptions are not signaled, they only are taken into account at signaling time.

Here are some illustrations of the previously described stack examples (cf.§.6.1) which show (1) how to control at the class level which exceptions will be propagated outside of methods defined on class *Stack*? (2) how to implement growing stacks by trapping exceptions on a new created *Stack* subclass.

```
Stack
    when: #(Overflow EmptyStack)
     do: ':exceptionObject | exceptionObject signal'
    when: ExceptionalEvent
     do: 'exceptObject: |  StackInternalException signal'

GrowingStack
    when: overflow
     do: ':anOverflow | self grow. anOverflow retry'
```

Our last example is a minimal vision of encapsulators [Pascoe 86], used to perform actions around the transmissions to encapsulated objects. The slot *encapsulated* is defined on the class *Encapsulator*.

```
Encapsulator
   when: DoesNotUnderstand
   do: 'e: | |result|
         <before actions>
         result <- encapsulated
                    perform: (e messageSelector)
                    withArguments: (e messageArgs)
         <after actions>
          e resumeWith: #supplyValue: with: result
```

7.3. Default handling and debugging.

Default handlers are methods defined on exceptions. They are invoked as shown in the top-level loop example (cf.§.7.1). Here is the most general default handler, defined on *ExceptionalEvent*, that can of course be parameterized on each exception. The method notify reports the exceptional event, its context an displays some propositions for interactive handling.

```
ExceptionalEvent methodFor:  defaultHandling!

handlesByDefault
     self notify.   ... interactive debugging entty point
     self retry.    ... Reenter the current top-level loop
```

Propositions [Moon&al83] are another idea to exploit exception hierarchies. Default handlers display all the proposition

Fig.5 : A Notifier taking into account propositions.

defined on and inherited by the current exception. A proposition is a couple of two method names, one to display a string and one execute a corresponding action. Propositions are stored for each exception in the slot named *propositions* defined on *ExceptionClass*. Here is (cf.fig.5) our adaptation of the standard *Smalltalk-80* notifier taking into account propositions and showing what happens when *DoesNotUnderstand* is signaled.

The hierarchy of exceptions induces a style of exception handling: all applications programs using the EHS will inherit the same basic debugging environment. The object-oriented representation of exception has many other applications related to debugging [Lieberman 87] [Dony 89].

8. Related Work.

Our basic representation of exceptions and our protocol for signaling are directly inspired from *Zetalisp*. We have added the meta-class *ExceptionClass* in order to define the method *signal* and the slot *propositions*. We have defined all protocols for handling on exception classes; in *Zetalisp,* all *lisp* control structures can be used within handlers to achieve various kind of non local moves, thus handling is not generic and various signaling primitives are to be used to signal either fatal or proceedable exceptions. Besides, there exists no way to attach handlers to classes.

[Borgida 86] [PP 89] or [Koenig&al90] are other object-oriented systems where exceptions are represented by classes, we do not have enough room to describe them. In most other EHS dedicated to OOLs, the ability to create and structure data is not yet exploited to manage exceptional event. Exception are not first-class objects, can neither be organized hierarchically nor own properties. Thus, we will now focus on signaling and handling possibilities.

Smalltalk-80 in its basic specification [Goldberg&al83] is the example of a language where handlers can only be attached to classes and are methods invoked by standard message sending. The system is simple (both to use and to implement) and efficient since it only uses message sending and method definition which are basic operations. We have outlined the interest and the drawbacks of this solution. Modularity is restricted by the impossibility of attaching dynamic handlers to expressions or statements.

A second category of languages consists in extensions of existing ones (for example *Clos*) implemented without modifications of, or additions to, the existing exception handling mechanisms, they do not provide any solutions to associate handlers with classes. Other languages also built on top of existing ones (such as *Loops* or *ObjVlisp*), chose a compromise: some exceptions relative to object manipulations are raised and handled as in *Smalltalk-80* , while the others are raised and handled by the original system. As the two systems are not connected, it is for example impossible to associate a handler for does-not-understand to an expression, and a handler for divide-by-zero to a class.

Objectworks Smalltalk v2.5 supplies both the original EHS describe throughout this paper and a new EHS allowing to associate handlers with expressions. The two systems have been connected in version 2.5; some exceptions (e.g. doesNotUnderstand) are still signaled in the old way by the virtual machine and the method *doesNotUnderstand* on class *Object* now convert the exception by signaling it with the new protocol. Anyway, the above described drawback of class handlers (static scope, cf. § 6.2) still applies, in our exemple, a class handler defined on class C1 would not be invoked.

Eiffel [Meyer 88] is the only system where handlers being associated with both instructions (methods bodies) and classes are taken into account by the same propagation algorithm. The semantic of propagation is nearly the same as in our system (although they are otherwise very different e.g. concerning the knowlede representation and the handling capabilities). When an exception occurs, a rescue clause (handler) associated to the current method is searched, if none exists a rescue clause is searched in the current class. If none exists the exception is propagated to the method caller. One main difference with our class-handlers is that, since rescue clauses trap all exceptions, a method rescue clause always has precedence over a class rescue clause even if the second one is more relevant to handle the exception.

9. Conclusion.

We have proposed an original exception handling system which synthesize the qualities of existing systems. It uses the object-oriented knowledge representation to solve many classical exception handling issues. It takes into account the specificities of object-oriented programming. It inherits the qualities of object-oriented systems, namely reusability and extendibility.

We have attempted to build a powerful and a reasonably simple system with two control structures for defining handlers, three basic methods for handling and three basic exception classes.

Exceptions are classes organized in a hierarchy. Slots can be used to parameterize exceptions. The hierarchy allows a single handler to catch any set of unexpectedly raised exceptions. Default handlers can be defined on exception classes, they are easy to localize and to parameterized. Handler can be associated with expressions and with classes. Exceptions are raised along the invocation chain. A class can thus control which exceptions will be propagated out of its methods, a method can handle lower level exceptions raised by inner method activations and provide context dependent answers. All handlers have a unique argument which is the instance of the created exception. Handling is a generic operation since it can only be performed by sending messages to this object or by signaling a new exception. Since handling capabilities rely on the signaled exception, a unique signaling primitive is provided.

The kernel exception classes and their associated protocols can be seen as well as a usable EHS or as a laboratory for designing some new ones. All methods of the system can be either hidden or redefined on new subclasses to design for example sub-systems in which resumption is impossible or with different rules for searching handlers[Levin77].

Aknowledgements

Thanks to Michel Bidoit for many discussions, to Jean-Pierre Briot, Pierre Cointe and Henry Lieberman for commenting preliminary versions of the paper, to Loic Lescaudron, Makoto Murata and Hayssam Saleh for various technical help.

References

[Borgida85] A.Borgida: Language Features for Flexible Handling of Exceptions in Information Systems.ACM Transactions on Database Systems, Vol. 10, No. 4, pp. 565-603, December 1985.

[Borgida86] A.Borgida : Exceptions in Object-Oriented Languages. ACM Sigplan Notices, Vol. 21, No. 10, pp. 107-119, October 1986.

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

[Caseau87] Y.Caseau : Etude et Réalisation d'un langage objet : LORE. Thèse de l'université Paris-Sud , Orsay, France, Novembre 1987.

[Christian82] F.Christian : Exception Handling and Software Fault Tolerance, IEEE Transactions on Computers, Vol. C-31, No. 6, pp. 531-540, June 1982.

[Cointe87] 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.

[Dony88] C.Dony: An Exception Handling System for an Object-Oriented Language.Procs of ECOOP'88, pp. 146-161 Oslo, Norway, Aug. 1988; Lectures Notes in Comp. Sci. 322.

[Dony89] 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.

[Goldberg&al83] A. Goldberg, D. Robson : SMALLTALK 80, the language and its implementation. Addison Wesley 1983.

[Goodenough75] J.B.Goodenough : Exception Handling: Issues and a Proposed Notation. Communication of the ACM, Vol. 18, No. 12, pp. 683-696, December 1975.

[Horowitz84] E. Horowitz: Fundamentals of Programming Languages. Springer Verlag, Berlin-Heidelberg, New York, Tokyo 1984.

[Ichbiah&al79] J.Ichbiah & al : Preliminary ADA Reference Manual. Rationale for the Design of the ADA Programming Language. Sigplan Notices Vol. 14, No. 6, June 1979.

[Knudsen87] J.L.Knudsen : Better Exception Handling in Block Structured Systems. IEEE Software, pp 40-49, May 1987.

[Koenig&al90] A. Koenig, B. Stroustrup : Exception Handling for C++. Usenix C++ conference Proceedings, pp. 149-176, San Francisco, USA, April 1990.

[Levin77] R.Levin : Program structures for exceptional condition handling. Ph.D. dissertation, Dept. Comp. Sci., Carnegie-Mellon University Pittsburg, June 1977.

[Lieberman87] H.Lieberman: Reversible Object-Oriented Interpreters. Proceedings of ECOOP'87, special issue if BIGRE No 54, pp 13-22, June 1987, Paris.

[Liskov79] B.Liskov, A.Snyder : Exception Handling in CLU. IEEE Transactions on Software Engineering, Vol. SE-5, No. 6, pp. 546-558, Nov 1979.

[Meyer 88] B.Meyer: Object-oriented software construction. Prentice-Hall, 1988.

[Mitchell&al77] J.G.Mitchell, W.Maybury, R.Sweet: MESA Language Manual.Xerox Research Center, Palo Alto, Calif., Mars 1979.

[Moon&al83] D. Moon, D. Weinreb : Signalling and Handling Conditions, LISP Machine Manual, MIT AI Lab., Cambridge, Massachussets, 1983.

[Nixon83] B.A.Nixon : A Taxis Compiler. Tech. Report 33, Comp. Sci. Dept., Univ. of Toronto, April 83.

[Pascoe 86] G.A.Pascoe: Encapsulators: A New Software Paradigm in Smalltalk-80. Proc. of OOPSLA'86, Special issue of Sigplan Notices, Vol. 21, No. 11, pp 341-346, November 1986.

[Pitman88] K.Pitman: Error/Condition Handling. Contribution to WG16. Revision 18.Propositions pour ISO-LISP. AFNOR, ISO/IEC JTC1/SC 22/WG 16N15, April 1988.

[PP 88] Objectworks for Smalltalk-80, ParcPlace Systems, 1988.

[PP 89] Objectworks for Smalltalk-80, version 2.5, Advanced User's Guide, Exception Handling. ParcPlace Systems, 1989.

[Yemini&al85] D.M.Berry, S.Yemini\ : A Modular Verifiable Exception-Handling Mechanism. ACM Transaction on Programming Languages and Systems, V0l 7, No. 2, pp 213-243 April 1985.