Smalltalk/V Exception Handling Proposal

Allen Wirfs-Brock
Version 0.1, October 14, 1992
Version 0.5, March 18, 1992

This document presents a proposed exception handling model for Smalltalk/V. The model is a synthesis of ideas from Christophe Dony ("Exception Handling and Object-orient Programming: Towards a Synthesis", OOPSLA'90) as implemented for Smalltalk/V by Hal Hildebrand, the Objectworks Smalltalk exception handling system, and original ideas. The primary goal of this proposal is to develop an exception model that includes a subset that is very simple for the casual Smalltalk programmer to understand and use, while at the same time supporting (or at least could be extended to support) the needs of the most sophisticated commercial application developer.

This document is (well, ultimately will be) in three parts. Part 1 describes the simple, casual user subset of the model. Part 2 describes additional features of the full model. In general this document is written in the form of a user's manual for the exception handling system.

> *Commentary on the design is presented using text like this paragraph. This commentary is intended to explain the particular design decisions that we made in developing this model. The commentary is not intended to be part of the end user documentation.*

Part 1: Exceptions — The Simple Story

Exceptions are unusual or unexpected events that can occur during the execution of a Smalltalk program. When one of these exceptional events occurs we normally want our program to take some special action. For example, if we are reading data from a file and unexpectedly encounter an end-of-file we may want to stop processing and display an error message.

Exception Classes

In Smalltalk, exceptions are represented as objects. Different kinds of exceptions are defined using Smalltalk classes. Each exception class defines a default action that will be performed upon occurrence of the exception.

The following table list some commonly used exception class. For each class the table describes what exceptional event is represented by the class and what default action is performed.

Error	any program error	open an error notifier
ArithmeticError	any error evaluating an arithmetic operator	inherited from Error
ZeroDivide	a attempt to divide by zero	inherited from ArithmeticError
FileError	any error associated with processing files	inherited from Error
ControlError	any error associated with evaluating blocks or sending messages	inherited from Error
DoesNotUnderstand	a message was sent to an object that did not define a corresponding method	open a notifier that allows the message send to be retried.
Notification	any unusual event that does not impair continued execution of the program	do nothing, continuing executing
Warning	an unusual event that the user should be informed about	Open a notifier that describes the event and allows the user to choose to continue execution.

The first major design alternative is whether the various exceptional conditions should be represented as different instances of a single universal exception class or whether each exception should be represented by a unique class within a hierarchy of exception classes. PPS uses the single class approach. Its advantages are fewer classes and potentially more flexibility in organizing the relationships between exceptions. (In theory, exceptions could be related using a lattice, not just a tree, but PPS does not do this.) The main disadvantage of this approaches is that all exceptions must have exactly the same protocol and the same encapsulated state. Another disadvantage is that the hierarchy of exceptions is not directly viewable using the browser. A separate class is required to represent the state of a signaled exception. Dony uses the multiple class approach. Classes represent exceptions that can be signaled and instances of the classes represent actual signaled exceptions. The relationship of the classes in the class hierarchy mirrors the priority hierarchy of the actual signals. The major advantage of this approach is that different types of exceptions can have unique message protocols and instance variable structures. Additionally, the set of available exceptions and their relationship scan be directly viewed using the browser. I choose the class hierarchy approach because of these advantages.

I have attempted to define an exception hierarchy that only includes classes that actually represent exceptions that a user might want to directly deal with. In particular, I did not include any abstract classes that implement handling alternatives such as proceedable nor not proceedable. One problem with that approach, is that the alternatives do not form a strict hierarchy. Some FileErrors may be proceedable while others are not. Similarly, a subset of the ArithmeticErrors might be proceedable. Dony solves this problem in his paper by using multiple inheritance. Hal implements this using a single inheritance hierarchy and copying behavior where multiple inheritance would be required. This results in a hierarchy where it looks like any exception is either proceedable or not proceedable but where in fact both attributes may apply to some exceptions. My solution is to eliminate these structural classes and make proceedable an attribute of each exception class.

Dony calls exceptions "Exceptional Events" and uses class names such as "FatalEvent" and even "Event". PPS uses the name Signal for its single class and "names" its instances things like ZeroDivideSignal. I don't like the Event nomenclature because the word "Event" shows up in many application areas and I don't want to preclude users having classes of this name. Signal is not an appropriate name for our classes because their instances are not things that can be signaled but the object that represents the occurrence of an actual, signaled exception. Because of this I decide it is most appropriate to 'call a spade, a spade". Our classes of exceptions are named "Exceptions". A secondary benefit of using names that are disjoint from both PPS and Dony is that it permits the development of compatibility packages that support those exception models.

Handling Exceptions

In some cases, a programmer may wish to do something other then the default action associated with an exception. This is accomplished by associating an *exception handler* with the execution of a Smalltalk block. For example:

```
[x / y] on: ZeroDivide do: [Transcript show: 'zero divide detected'.]
```

This expression causes its receiver (the block containing "x / y") to be evaluated by sending it the message value. If a ZeroDivide exception occurs while executing the block, the handler block (the argument to do:) is evaluated, causing a message to be written to the transcript.

All a programmer needs to learn to handle exceptions is the names of some exception classes and the on:do: message. PPS calls their equivalent message handle:do:.

Both PPS and Hildebrand require the argument to the do: keyword to be a block that takes a single argument. An object that captures information about the actual exception (exception object) is passed as the actual argument. My experience is that most exception handlers do not actually refer to the argument. In particular, handlers written by less sophisticated programmers seldom use the argument. For this reason I allow the handler block to be either a zero or one argument block. Simple handlers and simple programmers need not worry or learn about the argument.

An exception handler normally completes by returning the value of the handler block in place of the value of the receiver block. Note that the above example would return the transcript. If instead of displaying a message, we just wanted to return the value 0 when a division by zero occurred we would rewrite the above expressions as:

```
[x / y] on: ZeroDivide do: [0]
```

This might be used as follows:

```
fudgeFactor := [x / y] on: ZeroDivide do: [0].
```

If instead of returning a value we want to exit the current method we can place an explicit return within the handler block:

```
fudgeFactor := [x / y] on: Error do: [^'uncomputable'].
```

Note that in the last example, we specified Error as the exception to be handled instead of ZeroDivide. When we specify an exception to be handled we are really saying that we want to handle the named exception and also any exceptions that are subclasses of the named exception. Because ZeroDivide is a subclass of ArithmeticError which is a subclass of Error, an attempt to divide by zero, or any other error that occurs while evaluating x/y will cause the enclosing method to return the string 'uncomputable'.

I have tried to make simple blocks used as handlers do the "right thing". For non-proceedable exceptions, this is to return a value from the on:do: message. For proceedable exceptions it is to return a value to the signaler of the exception. Hildebrand requires that all handlers explicitly terminate by sending an appropriate message to the block's argument. Otherwise, an "unhandled exception" exception is raised. This is too hard for to learn and use for simple cases.

Sometimes an exception handler needs to obtain information about the specific exception that it is dealing with. This can be accomplished by using a single argument block as the exception handler:

```
[x / y] on: Error
        do: [:theException |
            Transcript show: theException description.
            ^'uncomputable'].
```

An exception object will be passed as the argument to the handler block. The exception object is an instance of the class of exception that actually occurred. In the above example, the exception object might be an instance of ZeroDivide, instead of an instance of Error. All exceptions respond to the message description by returning a string that describes the actual exception. This is used in the above example to display a message in the transcript.

The most common thing that a casual programmer needs to find out about an exception is a textual description of what happened. Other messages may be sent to the exception object ,including messages to explicitly control what happens when the handler exits. See Part 2 for details.

Cleaning-up After Ourselves

The occurrence of an exception normally causes Smalltalk to discard the current work in progress. Sometimes a method will do something that requires a subsequent action regardless of whether or not an exception occurs. A good example is a method that places a lock on an external file. As long as the lock is in place, other users cannot

access the file. Such a lock needs to be released even if an exception occurs. The following is an example of a method that could be written in class FileStream to implement such behavior:

```
whileLockedDo: aBlock
    "Lock the receiving file.  Process the argument block while the file is locked
        then unlock the file.  Return the value of the argument block. Be sure to
        release the lock if an exception occurs."
    self lock.  "set the lock"
    ^aBlock ensure: [self unlock "clear the loc"]
```

The message ensure: when sent to a block causes the receiver to be evaluated just as if the message value had been sent to the block. The value returned is the result of evaluating the receiver. The argument to the ensure: method is a block that is called the *clean-up block*. After evaluating the receiver, the clean-up block is also evaluated. The clean-up block is also automatically evaluated if for any reason the receiver block does not return normally. In particular, if an exception occurs while evaluating the receiver, the clean-up block will be executed. Executing an explicit return that goes outside of the receiver block is another situation that causes execution of the clean-up block. This is illustrated by the following example:

```
myFile whileLockedDo:
    [myFile atEnd
        ifTrue: [^nil]
        ifFalse: [myFile next]]
```

The clean-up block within whileLockedDo: will be executed even if the argument block encounters an end of file and explicitly returns nil.

PPS uses the particularly ugly selector, valueNowOrOnUnwindDo: for the ensure: operation. Correctly handling non-local returns from the receiver block requires some simple virtual machine extensions. Part 2 describes a less commonly used variant of the operations.

Signaling Exceptions

Most of the exceptions that programmers need to deal with are detected by code within the standard Smalltalk/V class. Occasionally, a programmer needs to write a new method that signals the occurrence of an exceptional condition. This is especially true if the programmer has also created new classes of exceptions. An exception is signaled by sending the message signal to the class that defines the exception. For example:

```
Error signal
```

creates an error exception. If there is an active handler that deals with the Error exception, it will be executed. Otherwise the default handler will be executed. It is often useful to provide a textual description when signaling an exception. This is accomplished by sending the message signal: to an Exception class:

```
Warning signal: 'the disk is almost full'
```

The argument string will be incorporated into the value returned when the message description is sent to the resulting exception object.

I have tried to make the most common cases very simple. Dony also uses the selector, signal, but things get rapidly more complicated. PPS uses the selectors, raise and raiseWithErrorString: for these operations.

Error exceptions normally do not return to the method that signaled the exception. However, the default behavior for Notification, Warning, and some other exceptions is to return from the signaling message after executing the active handler for the exception. This is accomplished by returning the value returned by the exception handler as the result of the signal or signal: method:

```
(Warning signal: 'yes or no?') = 'yes'
        ifTrue: [Transcript show: 'yes'].
```

Exceptions whose default behavior is to return from the signaling message are called resumable exceptions becuase the resume exectution rather than returning from the exception.

This section has described the basic functionality of Smalltalk/V exception handling. These functions should be adequate for most uses. The exception class also supports additional protocol that is useful in special circumstances.

Part 2: The Rest of the Story

Conditional Clean-up

The ensure: message is used to guareentee that a block of code will always be executed after the receiver block, whether or not the receiver block completes normally. Occationally, it is necessary to guarrentee that a block of code will execute only if another block *does not* complete normally. This is accomplished using the curtailed: message. For example:

Just like ensure:, the message curtailed: when sent to a block causes the receiver to be evaluated just as if the message value had been sent to the block. The value returned

is the result of evaluating the receiver. The argument to the curtailed: method is a block that is called the *clean-up block*. Curtailed: differs from ensure: in that the clean-up block is only evaluated if for any reason the receiver block does not return normally. In particular, if an exception occurs while evaluating the receiver, the clean-up block will be executed unless the exception resumes execution[1]. Executing an explicit return that goes outside of the receiver block is another situation that causes execution of the clean-up block.

Explicitly Exiting Handlers

A handler block normally completes by executing the final statement of the block. The value of the final statement is then used as the value returned by the exception handler. Exactly where control is returned with that value depends upon the *resumability* attribute of the exception. A *resumable* exception returns from the the message that signaled the exception. A *non-resumable* exception returns from the on:do: expression that created the handler blocks. For example the following expression returns 5 as the value of the on:do: message:

```
([Error signal] on: Exception do: [5]) "returns 5 here"
```

while the next expression returns 5 as the value of the signal message:

```
([Notification signal "returns 5 here"] on: Exception do: [5]).
```

Most exception classes specify whether they are always resumable or non-resumable; however, it is possible for an exception class to specify for each signaled exception whether or not it will be resumable.

Note that resumability is a an attribute of an exception not of the exception handler. If a handler block is invoked for a resumable exception, tha handler block will return to the signaler. If it is invoked for a non-resumable exception, it will return from its to:do: message. This can be seen in the above example.

An instance of an exception can be explicitly tested to determine whether it is resumable. This is accomplished by sending it the message isResumable. The following example, the exception handler either returns 5 to the signaler or 10 from the do:on: message depending where the exception class defines proceedable or non-proceedable exceptions.

```
[ someExceptionClass signal ] on: Exception
    do: [:theExceptionl
            theException isResumable ifTrue: [5] ifFalse: [10]]
```

Occassionally it is desirable to conclude processing of a handler block before reaching the final statement of the blocks. This can be accomplished in several ways by sending appropiate messages to the argument of a handler block. The message exit:

---

[1]This rule also applies to ensure: clean-up blocks.

causes its argument to be returned as if it was the value of the final statement of the handler blocks. The following two handlers have exactly the same behavior:

```
[Error signal] on: Exception do: [5] .

[Error signal] on: Exception do: [:theExceptionl theException exit: 5].
```

The most common use of exit: is in a conditional expression of a complex handler block. If the argument of a handler block is a resumable exception, the message resume: can be used in place of exit:. For a resumeable exception, resume: behaves exactly the same as exit: but it is an error to send resume: to a non-proceedable exception. Doing so causes a ControlError exception to be signaled.

Another alternative to exit: are the messages return and return:. These messages cause the handler block to return from its on:do: message. The value returned is the argument of return: or nil if return is used. When sent to a non-resumable exception return: has the same effect as exit: but when sent to a resumable exception control is forced to return from the on:do: message instead of from the signal message that created the exception.

Another way to exit an handler block is with the retry message. This message terminates the handler block and retrys the evalutation of the receiver of the on:do: block. Any ensure: or curtailed: clean-up blocks that we created by the original evaluation of the receiver block or by he handler block are executed before retrying the evaluation.

```
[^ x / y]
        on: ZeroDivide
        do: [:exl
                y := 0.00000001. " make the divisor very small but > 0"
                ex retry]
```

The message retryUsing: is similar to retry but instead of evaluationing the original receiver block, the argument to retryUsing: is evaluated in its place. For example:

```
[self doSomeTaskTheFastWay]
        on: LowMemory
        do: [:exl ex retryUsing: [self  doSomeTaskTheSpaceEfficientWay]]
```

Other Features that Need More Commentary

The message pass can be used inside a handler block to terminate the block and execute any enclosing handler blocks for the current exception. For example:

```
[[1.0 / 0] on: ZeroDivide do: [:e | Transcript show:'1 '. e pass]]
        on: Error
```

```
do: [:f | Transcript show:'2 '. ]
```

will cause the text "12" to appear in the transcript because the Z [self someAction]

```
on: Warning
do: [:e |
        self initialWarningAction.
        e pass. "let any enclosing handlers deal with it"
        self finalWarningAction]
```

Complex Signaling

```
ex := SomeException new.
ex establishExceptionState: foo.
resumptionValue := ex signal.
ex testState "if resumable"
```

Exception Environment of Handlers – Handler blocks execute in the exception environment of the on:do: message, not in the environment of the signaled exception:

```
[[[1.0 / 0]
    on: Warning do: [Transcript show: 'inner warning';cr]
    ]
        on: ZeroDivide do: [Warning signal: 'zdiv']
        ]  on: Warning do: [Transcript show: 'outer warning';cr]
```

causes "outer warning" to appear in the transcript.