Exceptional Power and Control

Juanita J. Ewing
Instantiations, Inc.

Derived from Smalltalk Report

In my last column, I discussed return values, and the use of specialized return objects. A return statement is only one mechanism that controls exit semantics and values. Another mechanism, an extremely powerful one, is exceptions. Exceptions provide flow control that crosses and encompasses methods.

The examples in this column are from the Smalltalk/V exception handling system. Objectworks\Smalltalk also has an exception handling system, so I will point out equivalent expressions during the discussion. We will also use the return object example from the last column as an example of how to add exceptions to an existing subsystem, including some architectural suggestions.

Simple Use of Exceptions

With an exception handling system, a developer can control how exceptional situations manifest themselves. In most Smalltalk systems, exceptions manifest themselves as errors, resulting in a walkback or error notifier. Exception handling gives developers the ability to control the manifestation of errors from low-level code, so they don't bubble up to the end user.

The basic premise of an exception handling system is simple: errors cause exceptions. Client code can ignore exceptions, which triggers the default action for the exception, or client code can handle the exception by performing a special action. Clients must designate which sections of code are protected from the default action of an exception.

To protect sections of code, protected code must be placed in a block, and sent the message on:do:. The first argument to the message is the exception the developer wishes to handle. The second argument is the handler block, which is the code to be executed in case of an error. The handler block optionally has a block argument, which is the exception that was raised. In Objectworks\Smalltalk, the equivalent exception handling capability is invoked by sending the message handle:do: to an exception.

The method isActive uses exception handling in a straight forward manner. It makes sure that file errors do not interfere with the test for the existence of the info file, by placing the test in protected block. It invokes protected execution with the message on:do:. The argument to the on:do: message specifies the exception FileError. The handler block contains the special action that is executed if a FileError is raised during execution of the protected block. If there are file errors accessing the info file, we assume the file does not exist, and exit from the method with a return value of false. In this method, the handler block does not have an argument, which is the

simpler form of the handler block.

```
isActive
   "Answer <true> if there is already an info file in the receiver's directory"

   ^[self infoFile exists]
      on: FileError
      do: [^false]
```

The on:do: statement in the isActive method handles the exception FileError and all of the exceptions derived from FileError. Unrelated exceptions are not handled in this statement. In the Macintosh implementation of Smalltalk/V, the FileError hierarchy looks like this:

```
FileError
        DirectoryNotFound
        EndOfFile
        FileDoesNotExist
        VolumeNotFound
```

These derived exceptions are specific kinds of file errors. In Smalltalk/V, exceptions are implemented as classes, so you can use standard browsing tools to examine and edit the exception hierarchy.

Example description

Our example from the last column used an operation that had several different return values, requiring the client to execute conditional code or perform a kind of case statement in order to use the result of the operation. We rearchitected the solution for our example, ending up with a specialized return object. The specialized return object could be queried to determine the success of the operation, and included more queries to determine if an exceptional condition had arisen.

Here is the description of our example operation:

- ¥ It might not succeed.
- ¥ The operation has a second chance of success - it can be retried with some input ignored.
- ¥ If the operation fails, it might be because of an internal error, or because an external function failed. For debugging purposes, it is desirable to distinguish between the two.
- ¥ Another effect of the operation is the creation of an OrderedCollection of strings containing result data from the operation.

The invocation of the operation using a specialized return object (simplified slightly from the previous column):

```
invokeOperation
      Invoke the operationWithPoorInterface. Return a collection of strings if the operation
   succeeded. If it failed return an empty collection.
```

```
| result |
result := self operationWithPoorInterface.
result wasSuccessful
        ifTrue: [self notifySuccess.
                ^result stringCollection].
result wasDataIgnored
        ifTrue: [self notifyDataIgnored.
                ^result stringCollection].
self notifyError: result errorMessage.
^OrderedCollection new
```

With the goal of simplifying the interface to the specialized return object, we rewrite this invocation using exceptions. Although the initial version of the invocation does not have as much capability as the original version, we will improve on the exception version of the invocation as this article progresses. Here is the simple initial version:

invokeOperation
Invoke the operationWithPoorInterface. Return a collection of strings if the operation succeeded. If it failed attempt a retry after user confirmation. Return an empty collection on failure.

```
| result |
[result := self operationWithPoorInterface]
        on: OperationWithPoorInterfaceError
        do: [: exception |
                self notifyError: exception errorMessage.
                ^OrderedCollection new].
self notifySuccess.
^result stringCollection
```

In this version the operation is performed while protected from errors, using the on:do: message. If no errors occur, we execute the code after the on:do: message, which notifies the user of success and returns the result. If an error does occur, we notify the user of an error.

Why is this architecture better?
The main difference between the example invocation with exceptions and without exceptions is the use of the on:do: message and the number and kind of messages sent to the specialized result object. The original invocation contained queries to the return object about errors. The new version does not contain queries about errors. The original specialized return object had information about two things: error conditions and operation results. With exception handling mechanisms, we can move the information about errors to the exception object. This partitioning of responsibilities results in more understandable and reusable code.

Though it is not so obvious by analyzing the client code, the developer has better control mechanisms with exception handling. The basic capabilities of the exception handling system

allows the developer to elegantly handle errors generated at a low level. This is extremely important for complex operations. Behind the original implementation there was special purpose code containing specialized calls to low level operations that prevents low level errors from bubbling up to the user. The specialized invocations are eliminated by using exception handling.

How do we specialized exceptions?
Because exceptions in Smalltalk/V are implemented as classes, it is easy to extend the exception hierarchy using the same mechanisms used for extending the class hierarchy. If you have a need for specialized exceptions, then you should create an extension of the exception hierarchy. It is convenient to root all related exceptions at a single exception. This allows clients to write simple code to catch all related exceptions.

Most developers create a set of exceptions for each subsystem. This simplifies the interface between subsystems by providing a consistent and extensible way to pass error and notifications between subsystem.

Most systems have different exceptions for different kinds of errors because the client needs to distinguish between kinds of errors. When you are designing your hierarchy, for example, you might want to group resumable exceptions together. After analyzing our example operation, we decide to use an exception hierarchy like this:

```
Error
        OperationWithPoorInterfaceError
                PoorInterfaceExternalError
                        PoorInterfaceFileError
                        PoorInterfaceResourceError
                PoorInterfaceInternalError
                        PoorInterfaceMissingInputDataError
                        PoorInterfaceUncomputableError
                        PoorInterfaceConflictingDataError
```

We want to distinguish between internal and external errors because the operation can be reattempted after an internal error. In this hierarchy, we make the distinction explicit by creating an exception hierarchy for each kind of error.

Why do we do we root our exception hierarchy at Error? One reason is that we want to inherit the appropriate behavior. One indicator of behavior is the default action of an exception. Here are the high level exceptions in the system, along with their default action:

```
    Exception - open a walkback
        Error - open a walkback
        Notification - no action
                Warning - open a warning message dialog
```

The errors we generate from our example are serious problems, not just warning or notifications. That makes Exception and Error potential derivation roots of our example exceptions because they have the appropriate default action: opening a walkback.

Of these two possibilities, we choose to derive our new exception from Error. We choose Error because it fits the standard way to catch all errors - an on:do: statement handling Error. The alternative is to catch all errors by handling Exception, but that combines catching errors and notifications. It is rare to want to treat notifications like errors!

Extending Exceptions

In addition to creating new exceptions, the Smalltalk/V exception system also has the capability of extending exceptions by adding behavior or state. It is good practice to limit extensions to your own exception classes, so that your extensions do not collide with modifications made by the vendor.

Let s return to our invocation of the OperationWithPoorInterface. The retry mechanism is convenient for allowing the end user to control this operation. Once we have determined the set of exceptions for our operation, we also want to implement a new message to determine if the operation can be retried. If the error is internal the end user is notified that he can retry the operation.

```
invokeOperation
    Invoke the operationWithPoorInterface. Return a collection of strings if the operation
succeeded. If it failed attempt a retry after user confirmation. Return an empty collection on
failure.

    | result |
    [result := self operationWithPoorInterface]
            on: OperationWithPoorInterfaceError
            do: [: exception |
                    (exception canRetryOperation and: [self canIgnoreData])
                            ifTrue: [self notifyRetryPossible: exception errorMessage]
                            ifFalse: [self notifyError: exception errorMessage].
                    ^OrderedCollection new].
    self notifySuccess.
    ^result stringCollection
```

The message sent to the exception to determine whether the operation can be retried, canRetryOperation, is a nonstandard message. Our specialized exception hierarchy must implement it.

We implement the message canRetryOperation at two different spots in our exception hierarchy. At the top, in the exception OperationWithPoorInterfaceError, we implement canRetryOperation to return false. For PoorInterfaceInternalError, we implement canRetryOperation to return true.

Developers can add state to exceptions, if necessary, by adding instance variables. The state inherited from Error includes an error message, but various other exceptions contain specialized information. For example, the exception MessageNotUnderstood has state for the message which

is not understood. In our example exception hierarchy, we could add state to the conflicting data error, PoorInterfaceConflictingDataError, to describe which data are conflicting.

How are exceptions generated?
Our example showed us how to handle exceptions. We also need to know how to generate exceptions at the appropriate times. In our original example, the specialized return object contained error information. When we use exceptions, we need to replace code that stuffed error information into the specialized return object by code that raises exceptions instead. Let s examine a code fragment that used the specialized return object:

```
externalError := self externalOperation.
externalError >0
        ifTrue: [aPoorInterfaceResult errorCode: externalError.
                ^aPoorInterfaceResult].
```

Instead of sending messages to the return object, we need to rework this code fragment to raise an exception. The default way to raise an exception is to send the message signal or signal: to an exception. Our reworked code looks like this:

```
externalError := self externalOperation.
externalError >0
        ifTrue: [PoorInterfaceExternalError signal: (self errorMessage: externalError)].
```

The signal message raises an exception. The signal: message raises an exception accompanied by a descriptive message. Other exception have specialized instance creation message appropriate for their extended state.

From one error, we can create another kind of error. To do this, we handle the first error, and from the handler block raise another error. In this code fragment, we catch a file error, and raise a specialized file error:

```
[fileStream := self createTemporaryFile]
        on: FileError
        do: [:exception |
                PoorInterfaceFileError signal: exception message]
```

Finer Control
There are a variety of ways to exit from an exception handler, each providing a different form of finer control. Exit mechanisms include resume, return, pass and retry. Some or all of these mechanisms are extremely useful with multiple exception handlers, but can also be useful with a single exception handler. All of these mechanisms are invoked by sending messages to the exception inside the exception block. Of these mechanisms, we will discuss retry and resume in detail.

The retry mechanism is used to re-evaluate the protected block, the receiver of the on:do: message. It is invoked by sending the exception the message retry. There is a variation of retry

that allows an alternate block of code to be evaluated. It is invoked with the message retryUsing: and takes the alternate block as it's argument. In Objectworks\Smalltalk, the retry mechanism is invoked with the message restart.

We again come back to our specialized return object example. Our original example included information describing whether the operation had been re-attempted. The client had no control over the re-attempt. With exceptions, we can improve the invocation of the operation by moving the retry control to client. With the retry mechanism incorporated, the invocation looks like this:

```
invokeOperation
    Invoke the operationWithPoorInterface. Return a collection of strings if the operation
succeeded. If it failed attempt a retry after user confirmation. Return an empty collection on
failure.

    | result |
    [result := self operationWithPoorInterface]
            on: OperationWithPoorInterfaceError
            do: [: exception |
                    (exception canRetryOperation and: [self canIgnoreData])
                            ifTrue: [self confirmIgnoreData
                                    ifTrue: [self ignoreData.
                                            exception retry]].
                     Can t retry
                    self notifyError: exception errorMessage.
                    ^OrderedCollection new].
    self notifySuccess.
    ^result stringCollection
```

If queries indicate data can be ignored, then the operation is retried by ending the message retry to the exception. We continue to make use of extensions to query the exception.

Here is an example using the retry mechanism from the Macintosh version of Smalltalk/V. One of the classes that manages memory, AbstractMemoryHeapPolicy, has a method that is used to allocate heap memory. If the allocation fails, indicated by the exception MacNotEnoughMemory, then a low memory action is performed to attempt to recover space and the allocation is retried.

```
AbstractMemoryHeapPolicy
do: aBlock requiringHeapBytes: estimatedHeapBytes
   "Evaluate <aBlock> after verifying that there is enough room on the heap to allocate
<estimatedHeapBytes>.  Perform lowHeapMemoryAction: if there isn't enough room.

Simplified for example."

   ^(self roomOnHeapFor: estimatedHeapBytes)
     ifTrue:
       [aBlock
```

```
        on: MacNotEnoughMemory
       do:
          [:ex |
           (self lowHeapMemoryAction: estimatedHeapBytes)
             ifTrue: [ex retry]]]
```

The other control mechanism I want to spend some time discussing is resume. Resume is a control mechanism that tells the exception handler to "keep going". Only resumable exceptions can be resumed.

```
MainWindow
close
   "Time to close the receiver. Check with the model, don't
   close if it doesn't want to."

   | allowClose |
   allowClose := true.
   [self triggerEvent: #aboutToClose]
      on: VetoAction
      do: [:ex | allowClose := false. ex resume].
   allowClose
      ifTrue:
         [self closeWindow]
```

In the close method, the aboutToClose event is sent to all objects that have registered an interest in the event. If any of the registered objects want to disallow closing, they signal a veto by raising the VetoAction exception. But, the processing shouldn t stop because of a veto. Each registered object must receive the aboutToClose event. The code is designed to handle this requirement: it notes the veto by setting the allowClose boolean to false, and proceeds to finishing informing registered objects about the intent to lose by resuming the protected block. After the protected block is complerely executed, informing the entire set of registered objects of the intent to close, the window is closed if no object has vetoed the close.

Which errors should you catch?

When handling errors, a good rule of thumb is to handle the most specific error that is appropriate. Specific handling is usually better than general handling, especially during development.

A common mistake is to write code that inappropriately handles the exception Error. More than one developer has been mystified by the cause of an exception, only to discover that their code catches all errors, including MessageNotUnderstood, a subclass of Error. In this case, generalized error handling covered up a coding mistake.

Ensured Execution

Another mechanism, built on exceptions, is the ability to ensure execution of some code. This mechanism requires placing protected code in a block, and the code whose execution must be

guaranteed in another block. Ensured execution will execute the ensured code no matter what happens, even if a return expression or an error terminates the protected block early.

This mechanism is particularly useful in cases that must reset state or that must be protected against inconsistencies. For example, this mechanism can ensure that a file will be closed after reading data from it. Smalltalk/V uses the message ensure:, which is sent to a block containing protected code and has the guarantee block as its argument. The Objectworks\Smalltalk equivalent is valueNowOrOnUnwindDo:.

This example is from the Macintosh version of Smalltalk/V. The method fill:withColor: uses ensured execution to make sure the background color is reset to its previous value. The background color will be reset, from the guarantee block, even if the erase operation, from the protected block, signals an error.

```
GraphicsTool
fill: geometricObject
withColor: fillColor
   "Fill the inside of a <geometricObject> with the given <fillColor>. The location of the receiver is not affected."

   | backgroundColor |
   backgroundColor := self backColor.
   [self backColor: fillColor.
   geometricObject eraseOn: self]
      ensure: [self backColor: backgroundColor]
```

Another related mechanism is one that guarantees execution of some code in case of an error. This code is executed only in the case of abnormal termination, such as with an error. The Smalltalk/V message to invoke this mechanism is ifCurtailed:. The Objectworks\Smalltalk equivalent is onUnwindDo:.

Conclusion

Exception handling is a powerful mechanism for controlling errors and notifications. Even simple applications can benefit from ensured execution and handling predefined exceptions. Complex applications can benefit from specialized exceptions. Each subsystem in the application should define specialized exceptions that are part of the public interface of that subsystem.