COMPUTING INTEGRITY
INCORPORATED
60 Belvedere Avenue
Point Richmond, CA 94801-4023
510.233.5400 Sales
510-233.5444 Support
510.233.5446 Facsimile

PROGRESS SOFTWARE BUSINESS ALLIANCE
ISV Partner

Exception and Condition Handling Classes – Version 0.1

8 May 2006

One of the problems faced by ABL programmers for many years is the lack of a true exception handling mechanism in the language. By "exception handling", we mean a mechanism by which a program encounters an error or unexpected condition then can communicate up the call stack with the purpose of:

1. Signaling that requested execution has not completed as expected;
2. Communicating any necessary information about the details of why this condition occurred; and
3. Providing a structure such that the condition is "handled" in a graceful fashion at an appropriate level in the code.

While exception handling support in many languages is heavily focused on "Exceptions[1]", i.e., circumstances which prevent continued execution in the normal flow of control, some languages generalize this mechanism to what some call condition handling[2]. The generalization to "conditions" broadens the potential communication to conditions of possible interest beyond those that one would normally consider errors or exceptions. Moreover, many conditions depend on context as to whether or not they should be considered an exception.

For example, given a routine intended to supply an address for a customer and the possibility that it may fail in obtaining that address. Even if it does fail, it can still return an address which is either null or consists of data such as "No Address Available". Whether this "condition" is "fatal" can depend on the context. Thus, in the context of a sales analysis report, the address may simply be providing context information and it is quite possible for the report to continue with all other processing to fulfill its purpose. In that context, treating this "condition" as an "exception", i.e., something that needs to interrupt the flow of control, would be undesirable since the purpose of the report can be fulfilled without this information. Conversely, if the context were printing invoices, one would probably not want to proceed with printing an invoice with a missing address. Or, if the sales analysis report involved a breakdown by state, that customer could not be included in the breakdown, but it could be included in a special category for unknown states and thus the report could complete normally.

Not only does the context determine whether a particular condition is an "error", but the correct place to handle that condition can vary, as can the method in which it is handled. Thus, in the case of the first sales analysis report referred to above, no action on the condition is necessary at the point where the condition is first noted. For the sales analysis report by state, some action might be required after the call to assign a special value. For the invoice program, the need is to abort current processing and return an "invoice aborted" condition to the next higher level as well as passing along the "address not found" condition so that it can be reported. For both sales analysis reports, it

[1] John Green, in a private communication, suggests a distinction between "error", a condition arising from the hardware or operating system platform, and "exception", a condition arising from the application.
[2] See, for example, the discussion at http://en.wikipedia.org/wiki/Exception_handling and http://www.franz.com/support/documentation/6.2/ansicl/section/conditio.htm

is probably desirable to retain the condition on some kind of "stack" so that an exception report can be added to the end of the report, where the information will be more apparent.

In many languages with exception handling mechanisms, exceptions are "typed". With strong typing, there is a compile time check to insure that the calling program and the called program agree on the types of messages that can be passed. This, like other forms of strong typing, is attractive because it ensures that the creating and consuming components agree on the signature, but has a potential disadvantage of requiring an excessive number of declarations for common exception conditions. This is potentially solved by allowing both strongly typed and weakly typed conditions where the later are used for things like DivideByZero that are ubiquitous.

Even with weak typing one is in a better position for evolving the software because one does not need to examine or modify a calling program just because some change is made in the way a called program generates its exceptions or the exact meaning of those exceptions, unless new exceptions are added or an exception changes from its original meaning. The alternative to typed exceptions tends to be something like the parsing of message content, which is very fragile.

Desiderata

Based on this discussion, we arrive at the following characteristics we would like to have in a full-featured exception and condition handling system:

1. A method by which a function can signal to a calling function that execution has not proceeded as expected or other notable conditions have occurred;
2. The ability to "type" these signals for reliable interpretation;
3. The ability to provide explanatory text with a signal;
4. The capability of having multiple signals active at the same time;
5. The potential for retaining signals on the "stack" for later action;
6. The ability to "cascade" signals up the call stack; and
7. The potential for making the handling of an exception imperative.

This last criterion deserves some additional discussion. One of the positive attributes of exception handling mechanisms in a number of languages is that, if a function generates an exception and that exception is not handled by the calling function, it is automatically propagated up the call stack until it is either handled or the top level function exits. This makes it certain that an exception is not accidentally and silently ignored. Of course, with a broader focused condition handling goal, some "exceptions" should not break the flow of control, but this simply means that code must be provided to "handle" those exceptions that does not break the flow of control.

Before moving on to the specifics of a mechanism for OOABL one might also consider aesthetics. Of course, aesthetics are individual so one person's preference may not be shared by another, but one of our concerns is that many exception handling mechanisms negatively impact the readability of code, at least with respect to the normal flow of control when exceptions do not occur. The classic example of this, of course, is the Java try[3] … catch … finally block. Instead of a clear readable sequence like:

```
MyObject:prepareContext( X, Y, Z).
MyObject:doTransformation( C ).
Q = MyObject::getResults().
```

[3] "Try" also has aesthetic objections because it seems like one has no confidence it the execution.

In which there is a simple, direct, and clear progression of events, one instead ends up with something that takes five times as many lines and has many references that have nothing to do with this main line of expected execution. E.g.,

```
try {
  MyObject:prepareContext( X, Y, Z).
} catch PrepartionException {
    Do something
}
try {
  MyObject:doTransformation( C ).
} catch TransformationException {
    Do something
}
try {
  Q = MyObject::getResults().
} catch ResultException {
    Do something
}
```

This is much harder to read than the core code on its own. Indeed, the most prominent thing one notices is "try, try, and try again"! And, this is far from the worst possible example since there is only one possible exception per try and no finally blocks. This obscurity contrasts with languages in which exception handling is managed by a block-structured construct such as:

```
begin:
  MyObject:prepareContext( X, Y, Z).
  MyObject:doTransformation().
  Q = MyObject::getResults().
  excecption:
    If exception = ...
     ...
  end exception.
end.
```

While this is an presentation that shows the flow of control more clearly, block structured exception handling may not be as much of an improvement as this simple example suggests depending on what the appropriate action is when an exception occurs. In the example as shown, if the response action for all three methods is to leave the block, push the exception to a higher level, or retry the block, this simple structure works fine. But, if the occurrence of an exception in the second method means that one should execute the second and third methods anyway, then that method would have to be contained in its own block, i.e.,

```
begin:
  MyObject:prepareContext( X, Y, Z).
  excecption:
    If exception = ...
     ...
  end exception.
begin:
  MyObject:doTransformation().
  Q = MyObject::getResults().
  excecption:
    If exception = ...
     ...
  end exception.
end.
```

It doesn't take a lot of additional blocks added solely for exception handling for the program to approach the obscurity of try … catch blocks. However, we still have a preference for a block structured implementation. In a full condition handling implementation, it could provide appropriate options about when a condition or exception was handled. One option would be to provide different signal levels, e.g., like the "severity" levels described below, and to provide alternate methods for responding by signal level, e.g., an "error" signal would immediately invoke the exception block, but a "warning" signal would not be processed until the end of the block. Another possible implementation might be a trigger-like approach, but this has the disadvantage of disconnecting the error handling code from the procedure.

Existing ABL "Exception Handling" Mechanisms.
There are basically two existing language features which relate to exception handling needs -- output parameters and `RETURN VALUE()`. Neither provides an adequate mechanism for several reasons.

Both methods do provide a means for providing some information back to the calling program, but both are flawed in similar ways, including:
1. Only a single value can be returned, so that multiple messages cannot be directly passed;
2. There is no mechanism for "typing" messages, other than by parsing the contents; and
3. There is no potential for "cascading" messages as they are passed up the call stack.

`RETURN VALUE()` is further flawed because there is no "reset" mechanism to insure that the value being tested was actually returned by the program just executed. These three points deserve some additional clarification.

Many exception handling mechanisms in other languages are also limited to returning a single exception per return, so this might not be considered an unacceptable limitation, but we feel that it is an unfortunate design principle if we are extending the requirement to condition handling. With condition handling there can be multiple non-fatal conditions from the same source and it is desirable to be able to cascade exceptions to higher levels. For example, consider a subprogram in a report that is intended to obtain address and telephone information about a customer. In the context of a report, it may be perfectly acceptable for this function to return a value such as "Unknown" or "Address not found" for both the address and the telephone because this information is not critical to the purpose of the report. To report that this has occurred via condition handling requires two separate condition types, one for address and one for telephone. Typical exception handling mechanisms are not well suited to this purpose, especially if a single procedure or method can generate multiple messages of varying severity.

The advantages of "typing" were discussed above. With output parameters and `RETURN VALUE()` one has an unfortunate choice of either passing back integer arguments which can be interpreted as typed error messages or passing back character data which can include explanatory messages and having to parse the content for the type. Parsing string data like this is fragile because any change in the called program may break the parsing. One could return multiple parameters, of course, but this is certainly not a very attractive solution.

When a program receives an exception from a called program, in some cases, it is able to handle the condition locally, but in other cases, it must pass the information farther up the call stack. For example, consider a function in which a top level function selected a set of invoices to print and that function calls another function to print an individual invoice and that function in turn calls a function to obtain customer name and address information. In this context, if the last function cannot obtain the address, it might be considered a fatal error for the invoice print function since one typically does not want to print an invoice with missing or incorrect information. Thus, the invoice print function needs to pass back to the calling program both the information that the particular invoice print failed and that the reason for the failure was the inability to obtain the name and address information. To do this with full flexibility requires both the ability to pass multiple messages and to cascade a message from one program to another.

One aspect of exception handling mechanisms in other languages that we can't fully replicate in OOABL without a change to the language is the "forced" nature of an exception. I.e., if program A calls program B calls program C and program C posts an exception, the exception is propagated back to program B and, if not handled by program B will propagate back to program A. If not handled in program A, the exception will result in a runtime fault. Obviously, this means that one would like to include a general exception handler in the top level program to insure at the least a graceful termination and logging of the error, but the important quality here is that the exception cannot be posted and then simply overlooked. Nothing but good programming practice is going to provide us with this assurance in OOABL until an internal exception handling mechanism is implemented.

Characteristics of CI Implementation

In the sample code provided with this document, we are providing two primary classes for managing the exception stack – ExceptionStack.cls and SimpleExceptionStack.cls. As indicated by the names, ExceptionStack.cls is the richer implementation, but requires more context, including a database table for standard messages. To use either class, it is instantiated as a pseudo-singleton as is described in a previous publication and then NEWed in each program in which it will be used. I.e., ExceptionStack.cls is actually a façade class and the actual implementation is found in ExceptionStackImpl.cls, which is instantiated as a singleton. The same is true of SimpleExceptionStack.cls and SimpleExceptionStackImpl.cls.

Both classes utilize the concept of exception severity classified into 5 levels:

ERROR	An indication that normal processing cannot continue. An example might be encountering an unexpected divide by zero. The number to indicate ERROR is 1.
WARNING	An indication of a condition that indicates a clear problem, but one that may not be fatal. An example might be a missing address for a customer. The number to indicate WARNING is 2.
CAUTION	An indication of a condition that may indicate a problem or something unexpected, but which doesn't appear to impact current processing. An example might be encountered a price of zero. The number to indicate CAUTION is 3.
NOTE	An indication of a condition of possible interest, but not necessarily indicating an error. An example might be a missing social security number in a context in which those individuals in foreign countries would legitimately not have social security numbers. The number to indicate NOTE is 4.
DEBUG	A trace or debug message not related to normal processing. The number to indicate DEBUG is 5.

These could be easily renamed, reduced, expanded, etc. to the preference of the individual development group. This hierarchy of severity levels is intended to allow filtering of conditions based on context. E.g., ERROR should always be handled, but NOTE might be optionally suppressed, e.g., when running a report repeatedly.

For ExceptionStack.cls, there is a database table tb_AppMessage that contains standard Exception messages filed by Exception number and language. Languages are indicated by three letter ISO codes, e.g., ENU for English. Messages in this Standard Exception table may have substitution markers (&1, &2, &3, etc.) which will be filled in with arguments when the Exception is posted. E.g., "Cannot access &1 record at this time; locked by user &2" might be posted with the arguments Customer and Joe and it would become "Cannot access Customer Record at this time; locked by user Joe". If a message is posted with an Exception number not in the table, then the text supplied in the arguments is posted as the Exception message. SimpleExceptionClass does not have the database table and so simply uses the message passed to it. The two classes are identical in all other respects and have the same interfaces. The database table provides stronger message typing and the option to internationalize the messages.

For both classes the method for adding messages to the push down stack is pushException(), which takes three input parameters – the Exception number, the severity in the form of a number, and a string containing either the comma delimited list of arguments or the text of the Exception message. Merely pushing down an Exception message does not cause a return to the next higher level on the program stack so multiple messages can be pushed down, either because processing can continue or because it is desirable to provide more than a single message about a particular event. The program name calling the pushException() method is also recorded. Exception numbers are not required, but are highly recommended for exception typing.

Both classes provide a trace() method which takes a single input parameter which is the message. An Exception number of -1 and a severity of 5, i.e., DEBUG, is assumed. Trace and debug messages can also be posted via pushException(), but the trace() method is provided as a convenience.

The method provided for retrieving the top message on the stack is popException(N). The argument specifies the minimum severity level to include. Messages are searched in LIFO stack order, but only messages which have a severity less than or equal to N will be returned. I.e., popException(1) will return only exceptions of type ERROR while popException(4) will include all exceptions except DEBUG/trace() exceptions. popException() returns a formatted string that begins with the severity formatted to a standard 9 character width[4] and the message. If the exception number is 1 or greater, the message will be followed by the exception number in parentheses. If the exception number is -1, i.e., generated by the trace() method, "[TRACE]" will be added after the message. If the program is in debug mode at the time the message is retrieved, the program name from which the exception was generated will be added. The exception that is returned is deleted from the stack.

One can also query the exception stack with numExceptions(N), which will return the count of exceptions of level N or below, i.e., numExceptions (1) will return a count of ERRORs, which might be a test used by a low level routine which wanted to allow any less severe exceptions to propagate to a higher level in the call stack. Also, getExceptionNo(N, I) will return the exception number of the Ith exception of level N or below. Thus, if a function wanted to check whether a called program

[4] This formatting to a 9 character width provides a consistent left edge for the message body in report lists.

had created exception number 302, because that was one exception that it knew required special handling, then it could check for this exception with:

```
do lin_Index = 1 to lob_MyExceptionStack:numExceptions( 4 ):
   if lob_MyExceptionStack:getExceptionNo( 4, lin_Index ) EQ 302
   then do:
       special processing
  end.
end.
```

getExceptionNo() does not pop the exception from the stack. dropException(N) will remove the same exception which would have been returned by popException(N), but no facility is currently provided for deleting an exception from other than the top of the stack, although this is not technically difficult if it is demonstrated that the need exists.

Additional methods include:

clearExceptions()	Removes all exceptions of all levels from the stack.
setDebug()	Puts the object in debug mode.
unsetDebug()	Takes the object out of debug mode.
inDebug()	Returns a true or false to indicate the current debug state.

The toLongString(N) method returns a longchar consisting of all exception messages of severity level N or less. These messages are formatted according to the same rules as for popException() and they are separated by a Control-A separator. This method is intended to provide a method for presenting messages in a browser or sending them to a log file. E.g.,

```
llc_ExceptionLog = lob_ExceptionStack:toLongString( 9 ).
output to "ExceptionLog.txt".
do lin_Entry = 1 to num-entries( llc_ExceptionLog, chr(1) ):
  put unformatted entry( lin_Entry, llc_ExceptionLog, chr(1) ) skip.
end.
output close.
```

The writeExternal() method returns a longchar which consists of XML representing the complete data member state of the object. This method is intended for serialization over possible remote connections.

Recommended Usage

These classes have been designed so that it is possible to begin using them in an existing application, either OO in structure or not, with minimum difficulty. All that is required is to instantiate the object at the highest level in the application and then to instantiate an instance in any program or class in which one desires to use this functionality. Because we have no “forced” exception mechanism in the current ABL, one can’t insure that an exception will propagate to higher levels if not handled, but some fairly simple good programming practice can approach this capability.

One should first provide an overall exception handler high in the call stack, e.g., after the return of any menu item. This exception handler should notice at least all residual, unhandled exceptions of severity 4 or less. A switch might be provided as to whether DEBUG/trace() exceptions should be logged, discarded, or retained. One might decide to exit the application if there are any unhandled ERROR conditions, but it not necessary. But, some mechanism should be provided in any case to alert the user to the existence of the conditions and a method provided by which they can be recorded.

Secondly, one should provide an appropriate general error handler in the top level of the function in which one wishes to implement this new functionality. The nature of this handler will depend on the type of function, but should be quite generic by function type. For example, in a report program, one will probably want to conclude with a handler which will display all residual exceptions on a separate page at the end of the report where it can be easily found. Alternatively, one could have other patterns of response such as e-mailing all level 1 errors to a support technician for investigation as possible programming bugs and putting all level 5 messages into a file with a standard name.

It is strongly recommended that development using these classes implement the more complex form and create error messages in the database table for all exceptions. Even if current deployment is all in one language with no plans for expansion, this approach provides stronger typing than merely relying on local values for exception numbers. A structure is also provided in the database table for an extended discussion of the error.

Use within a function is a simple matter of including a NEW of the class in the top of each procedure or in the constructor of any object that needs to either create or handle exceptions, along with a delete of that object in the final procedure or destructor. Whenever an exception or special condition occurs, simply insert a pushException() and decide whether or not an immediate return is required.

In the calling component, different handling styles will be required depending on the kinds of possible exceptions that are possilble. In a sub-component with no UI, one might simply have code such as:

```
SomeObject:doSomething().
if ExceptionStack:numExceptions(4) GT 0 then return.
```

This will pass the exception up the call stack for handling at a higher level. If this return implies some additional condition, such the invoice printing example discussed previously, this might be expanded to:

```
MyCustomer:getAddress().
if ExceptionStack:numExceptions(2) GT 0
then do:
   ExceptionStack:pushException( 1001, 2, “Invoice,Missing Address” ).
   return.
end.
```

Which will pass the missing address exception details to the higher routine and add a new exception that might use a message template of “&1 printing has failed due to &2”, i.e., producing “Invoice printing has failed due to Missing Address” in this case. Alternatively, one might imitate the block structure with code such as:

```
MyCustomer:getShipToAddress().
MyCustomer:getBillToAddress().
MyCustomer:getTelephone().
if ExceptionStack:numExceptions(2) GT 0
then do:
   mch_Message = ‘Invoice,’ + string(imin_InvoiceNum) + ‘”’.
   ExceptionStack:pushException( 1005, 2, mch_Message ).
   return.
end.
```

Where the message template was a simple “Printing of &1 number &2 aborted.” and the calling program might then list one or more messages about missing information following this message.

Because there is no forced exception feature, it is important during implementation that calling and called functions are kept in sync with respect to possible exception conditions. The use of top level general purpose exception handlers at the menu level and at the top level of any function can essentially eliminate the risk of exceptions going unnoticed, though naturally it can’t ensure that the exception will be handled in the way it should and at the level it should. The greatest difficulty one is likely to encounter retrofitting this approach to existing code is in adding this style of exception handling to common shared routines since all functions which use those routines will need to be updated.

Conclusion

While the exception and condition handling classes presented here do not compensate for the lack of a true exception handling facility in the language, they have an advantage in providing a more generalized facility than is provided by most exception handling mechanisms and thus may prove useful even if true exception handling is later added to ABL. In particular, if the new facility that gets added is a simple exception handling structure, then it may be desirable to continue to use this code to manage the overall stack and to simply add a “ThereIsSomethingToNotice” exception to trigger the calling function to examine the condition stack.