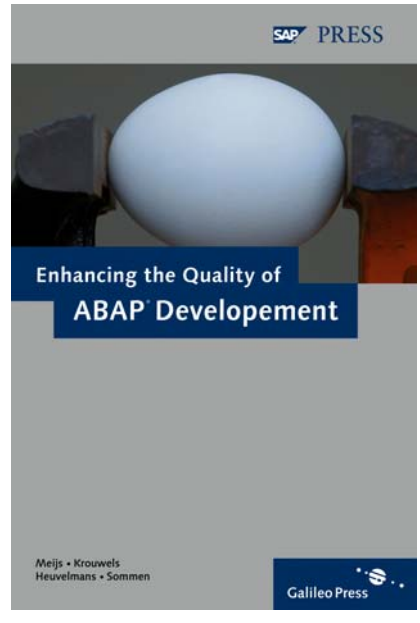
Ben Meijs, Albert Krouwels,
Wouter Heuvelmans, Ron Sommen

Enhancing the Quality of ABAP® Development



SAP PRESS

Contents

5 Exceptions and Error Handling

One of the best ways to make a program behave more predictably, and thereby, enhance a program's technical quality, is by implementing proper exception handling. Simply put, good exception handling will result in fewer errors. And, because one of the goals of exception handling is to make it easier for the user to respond to any outstanding errors, it will have a positive effect on user acceptance as well.

We broadly define an *exception* to mean "any deviation from the normal program flow."[1] Consider, for example, an implicitly generated returncode not equal to zero that is returned by a `SELECT` statement, or an unexpected value of a variable that is used in an IF condition. You should note that an exception is not the same thing as a coding error. *An exception is an unanticipated situation or event encountered during the execution of program logic.*

Exception handling is the way in which deviations are detected, caught, and handled. Dealing with exceptions is directly related to the two previously discussed main topics—correctness and stability—because both of these software requirements are directly affected by the following results of sloppy exception handling: runtime errors (program dumps), obscure error messages, incorrect data on lists or screens, and even inconsistent database tables. To a certain extent, you could view adding elegant exception handling as adding the finishing touch to a program in order to support its correctness and stability. You could even argue that exception handling is actually integral to ensuring technical quality.

5.1 The Importance of Exception Handling

It's hard to predict and anticipate all the possible circumstances in which a program will be used. Which exceptions are actually considered often depends directly on the context in which a program is used. It's quite difficult to predict all the exceptional situations that may arise. To a certain extent, a program's context will be clear, and so will the logic that is applied and the data that is processed. However, there's always a point at which the developer's (and other people's) knowledge about the context ends, and particular exceptions need to be taken into account. Ignoring these types of situations is not an option. Therefore, usually, a developer must deal with the typical details of programming techniques in general and the ABAP programming language in particular. And most often, these details aren't explicitly included; for example, in the functional specifications for a new program.

1 Note that an exception is defined more strictly in the second volume of Horst Keller, Joachim Jacobitz, *ABAP Objects: The Official Reference*, SAP PRESS 2003.

On the other hand, even without knowing the context in which a program is used, it's not really that difficult to address each imaginable deviation from the normal program flow. All that is required is a proper and unbiased look at the program code (which makes exception handling an excellent topic to be checked during a peer review (see Section 9.1). Moreover, the ABAP language provides all the necessary means to conduct this kind of check—to raise exceptions, intercept them, and take the appropriate action when necessary, in current releases and those prior to Web AS 6.20.

And yet, it's well known that exception handling in most customized ABAP developments is incomplete. We can think of various reasons for this:

1. **Because the programmer doesn't know how to implement exception handling.**
   Imagine, for example, that a piece of program code deep down the *Call Stack* of a program causes an error: a subroutine calls a function module; this function module, in turn, calls another function module; and this last function module generates an exception. The question then is: Where should the exception be handled? If it's better to wait to give an error message until the main line of the program is reached again, all the context information about the error must also be returned through the Call Stack.
2. **Because the assumptions of initial development aren't expected to become invalid.**
   Customized developments are often made to rely on assumptions that, according to the developer, make additional exception handling superfluous. As such, this may be true. However, this ignores the fact that maintenance may have to be applied later. At the time of initial development, the assumptions that are made usually cannot be questioned ("This simply can't go wrong. We have only two material types."). However, this will make the program more fragile. Unfortunately, this usually doesn't become obvious until a small change is implemented (for example, related to the introduction of a new material type). Particularly if the change itself comprises only a few lines of code, no one will consider it justifiable to reinvestigate all the assumptions made during the initial development—this would take much more time than applying the actual change. Nevertheless, even a small change can introduce exceptions that didn't exist. Therefore, it isn't that the assumptions are invalid when they're first made; the problem is that they can *become* invalid once the program is changed.
3. **Because it takes too much time.**
   Like testing and documenting developments, implementing exception handling is hardly the most exciting of all development activities; however, it can

require a large amount of the total development time of a program. And, if there are many exceptions to consider, it's easy to predict the following all too common scenario: every exception that doesn't appear to be essential or critical, or whose exception handling seems difficult to implement, will be ignored. For programmers, who may look for ways to avoid this activity and get away with it, skipping exception handling is a way to deliver a program according to plan and budget.

In this chapter, we'll address these basic dilemmas. We'll discuss which exceptions can be used. We'll explain the types of exception handling that are less frequently applied but that can nevertheless be useful for customized ABAP programs. And finally, we'll show you how to avoid the Call Stack dilemma by using the so-called *class-based exception handling* (that is, exception handling using ABAP Objects).

In Section 5.2, we'll first provide you with an overview of the basics of exception handling in an ABAP environment. Next, in Section 5.3, we'll present the main characteristics of class-based exception handling, and show you how to combine class-based with existing exception handling. In Section 5.4, we summarize the most important conclusions.

5.2 Implementation of Exception Handling

Exception handling distinguishes among three different elements (see Figure 5.1):

- Generating or raising the exception event
- Noticing or intercepting the exception
- Acting on the exception

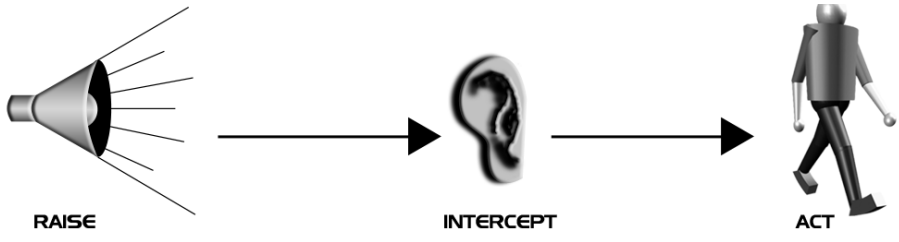


Figure 5.1 Raise—Intercept—Act

The first step in exception handling is raising the exception, that is, giving a signal that something has gone wrong. On the one hand, there are *implicit exceptions*; for example, a returncode that is not equal to zero when using an ABAP command such as `SELECT`, or a runtime exception such as divide-by-zero. On the

other hand, *explicit exceptions* are the ones caused deliberately by the program logic itself; for example, by setting a specific flag, or by using the `RAISE EXCEPTION TYPE <exception class>` command.

Independently of how an exception is raised, it must then be explicitly *recognized* (or detected, caught, intercepted) by the program code before anything can be done with it. Lastly, of course, the appropriate *action* must be taken. For example, consider sending a message to the end user. To summarize these steps, think of them as *Raise, Intercept,* and *Act*.

This section discusses the standard possibilities for implementing exception handling in an ABAP environment. In Section 5.2.1, we'll review the available options for generating and intercepting the exception *signal*. Next, in Section 5.2.2, we'll discuss the most important ways of implementing the exception handling.

5.2.1 Generating and Intercepting an Exception Signal

In this section, we'll give you an overview of how to intercept the following types of exceptions: a returncode; unexpected data in `IF` and `CASE` statements; an interface parameter of a subroutine; an exception generated by a function module; a class-based exception; and a runtime error. For the explicit types of exceptions, we'll also show you how each of them is generated.

Returncode

The first exception signal that each ABAP developer is likely to think of is a returncode that is not equal to zero (`sy-subrc` <> *0*). The system field *sy-subrc* is used by many ABAP commands to indicate whether the execution of the command has been successful: all SQL commands; internal table commands such as `READ` and `LOOP AT <itab>`; and file-handling commands such as `OPEN DATASET`. However, even if programmers know this, they often don't explicitly check the returned value, especially if they expect that nothing will go wrong.

Note that not every returncode indicates an exception. In most situations, such as when reading database tables or internal tables, a returncode not equal to zero represents an exception, even if the result is not immediately disastrous. But, sometimes, a returncode is checked as part of the program logic. To see the difference, consider the following two pieces of ABAP code. The first piece of code is an example of the use of a returncode that represents a genuine exception:

```
  OPEN DATASET tp_filename FOR INPUT IN TEXT MODE ENCODING DEFAULT.
  IF sy-subrc <> 0.
* Exception Handling here--
  ENDIF.
```

The second piece of code is an example of a returncode that does not point at an exceptional situation:

```
  DO.
    READ DATASET tp_filename INTO tp_string.
    IF sy-subrc <> 0.
* This is no exception but a check whether an end-of-file has been reached
      EXIT.
    ENDIF.
  ENDDO.

  CLOSE DATASET tp_filename.
```

The way in which the returncode is used in the first example represents an exception because the file is expected to be available. Even if the exception was foreseen, not handling it will lead to a dump as soon as the subsequent `READ DATASET` statement is executed. The second returncode does not represent an exception: it's just a way of establishing an exit-condition for the `DO` loop for processing the file in order to determine if the end of the file has been reached.

Unexpected Data in Conditions

ABAP programs are often based on assumptions about values of data. The statements `IF` and `CASE` are examples of statements that determine the flow of a program on the basis of expected values. For example, consider a program whose flow is determined by the value of a material type, and that only two material types are expected: *RAW* (for raw materials) and *SEMI* (for semi-finished products). In terms of its exception handling, an incomplete `CASE` statement would look like this:

```
CASE tp_materialtype.
  WHEN co_materialtype_raw.
* Handling of material type RAW starts here
  WHEN co_materialtype_semi.
* Handling of material type SEMI starts here
ENDCASE.
```

Consider what would happen if variable *tp_materialtype* contained a value other than `RAW` or `SEMI`. (The assumption that only these two values would be processed by the program was valid during initial development, but it is no longer valid.) This situation is not anticipated in the code. Without knowing more about the context of the code, it's nevertheless fairly easy to discern that the result of the ABAP program could become unpredictable. Perhaps, the rest of the ABAP code will proceed with incorrect data, or a runtime error will occur. We recommend ensuring that the exception is always caught, in this particular case, by adding a `WHEN OTHERS` condition, as follows:

```
CASE tp_materialtype.
  WHEN co_materialtype_raw.
* Handling of material type RAW starts here
  WHEN co_materialtype_semi.
* Handling of material type SEMI starts here
  WHEN OTHERS.
* The exception is caught here.
ENDCASE.
```

By inserting a `WHEN OTHERS` condition in a `CASE` statement, unforeseen situations are always recognized and exceptions can be handled accordingly (the same is true for `IF` statements).

Interface Parameters of a Procedure

Explicit interface parameters of procedures (that is, functions, methods, or subroutines) are often used to indicate to the calling ABAP code that an exception has occurred within the procedure. There are two possibilities: using `EXCEPTIONS` for functions and methods, and using `EXPORTING`, `CHANGING`, and even `TABLES` parameters to return results. An example of the second option is the `RETURN` parameter of BAPI function modules. This parameter, which has the structure `BAPIRETURN`, contains the result of the BAPI. The same technique is sometimes used in the interface parameters of routines. The following code is for a subroutine call:

```
PERFORM routine1 USING tp_input
              CHANGING tp_result_ok.
IF tp_result_ok = abap_true.
*
ELSE.
* Exception handling
ENDIF.

FORM routine1 USING  utp_input     TYPE c
           CHANGING  ctp_result_ok TYPE boolean.
* The specific function logic is executed here. As a result a
* corresponding returncode is set.
* ...
  IF sy-subrc NE 0.
    ctp_result_ok = cofalse.
  ELSE.
    ctp_result_ok = co_true.
  ENDIF.
ENDFORM.
```

If an exception occurs during the execution of `FORM routine1`, it isn't handled there immediately. Instead, it's reported back to the calling piece of code via the

interface parameter ctp_result_ok. So, immediately after executing the PERFORM statement, the value of the interface parameter will indicate whether an exception has occurred.

Passing on exceptions to a higher level in the Call Stack allows you to separate the code that takes action on an exception from the code that merely checks whether an exception has occurred.

Traditional Exceptions of a Function Module

For function modules, a separate part of their interface (the EXCEPTIONS parameters) is reserved for passing on information about exception situations. The associated kind of exception is deliberately raised in the function module (or method) by executing the RAISE <exception> command. The following code shows an example of a corresponding function call:

```
CALL FUNCTION '/CTAC/FM_WITH_TRAD_EXCEPTIONS'
  EXPORTING
    itp_input        = tp_input_fm
  IMPORTING
    etp_export       = tp_output_fm
  EXCEPTIONS
    not_found        = 1
    OTHERS           = 9.

CASE sy-subrc.
  WHEN 0.
* Everything is OK...
  WHEN 1.
* Action needed here...
  WHEN 9.
* Action needed here...
ENDCASE.
```

If a NOT_FOUND exception is raised by function /ctac/fm_with_trad_exceptions, system field sy-subrc is automatically made to contain value 1. This returncode must be checked by the calling ABAP code directly after the CALL FUNCTION statement.

Class-Based Exceptions

The second type of exception that is raised deliberately is the exception that is raised via using the RAISE EXCEPTION TYPE <exception class> command. This is part of the implementation of the class-based exception handling concept available as of Release 6.20 of the SAP Web Application Server (Web AS). After the exception has been generated, it is intercepted within a TRY-ENDTRY block with the command CATCH. The exception itself is an instance of a global exception class. We have added a basic code example:

```
TRY.
    SELECT SINGLE * FROM mara INTO wa_mara
           WHERE matnr = pa_matnr.
* ...
    IF sy-subrc NE 0.
      RAISE EXCEPTION TYPE  cx_sql_exception.
    ENDIF.
* ...
  CATCH cx_sql_exception.
* Include exception handling here
  CLEANUP.
* This is always executed whenever an exception occurs
    CLEAR wa_mara.
ENDTRY.
```

Runtime Errors

Runtime errors such as divide-by-zero errors or type-conflicts are clear signals of exception situations that are implicitly generated by the runtime environment. For example, let's look at the following code:

```
TRY.
   tp_average = tp_total / tp_count.

  CATCH cx_sy_zerodivide.
* Include exception handling here
ENDTRY.
```

If this code is processed and variable `tp_count` somehow contains a value of zero, a runtime error will raise an exception CX_SY_ZERODIVIDE. The exception can then be intercepted by using the `CATCH` statement before it can do any harm.

Note that some runtime errors can be intercepted (or caught) using either the `CATCH` command (up to Release 4.6C) or the combination of a `TRY` and `CATCH` command (as of Web AS Release 6.20). However, it is not possible to catch all runtime exceptions (for more information, see Chapter 4).

5.2.2 Implementing the Actual Exception Handling

After an exception signal has been raised and intercepted, you must act on it. You cannot simply intercept a runtime exception to prevent a dump, and then take no further action. The same is true for a returncode or any other exception signal. For example, the following ABAP code and added comment lines make little sense, but are nevertheless sometimes found in customized ABAP programs:

```
READ ta_itab INTO wa_itab WITH KEY keyfield1 = tp_value BINARY SEARCH.
IF sy-subrc NE 0.
* Now I don't know what to do. I presume this will never happen
* so this explains why no more actions were taken.
ENDIF.
```

So, doing nothing is not an option. On the other hand, you may need general guidelines for situations where you don't expect an exception to happen, but nevertheless must consider the possibility. In such cases, we recommend that you stop the program immediately: this at least clearly indicates that something went wrong. Letting the program go its own unpredictable way is always worse.

In this section, we address the proper implementation of the steps to be taken after an exception has been recognized. The main criteria that determine these steps are:

- Should the program send a message?
- Should the program execute cleanup actions?
- Should the program ask for a correction of data?
- Should the program continue after the exception has been handled?

These criteria are shown in Figure 5.2:

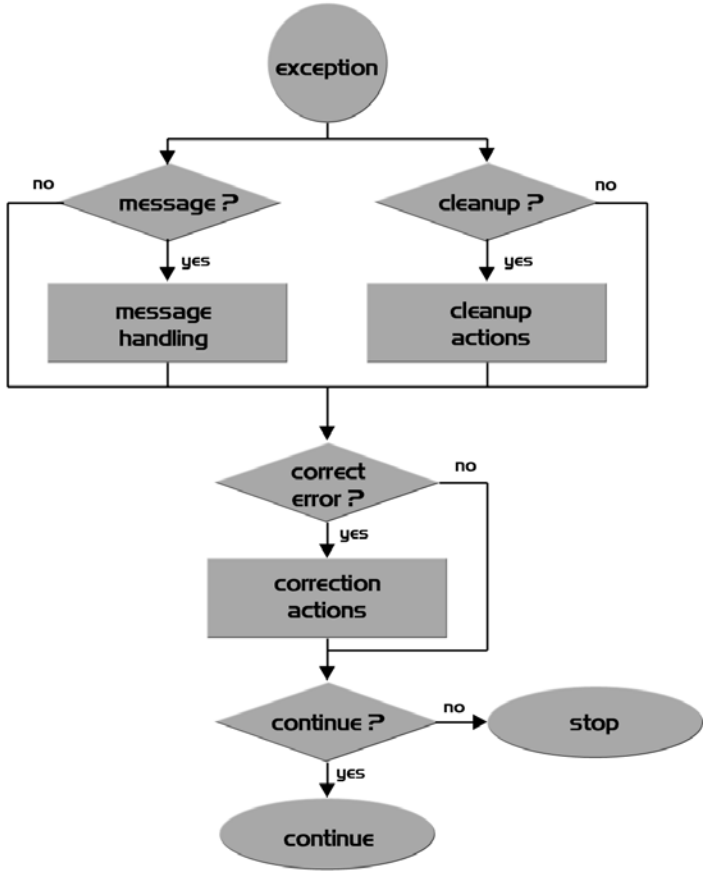


Figure 5.2 Exception Handling Flow

Let's first discuss some more details of the general exception handling logic: message handling and other important actions, mainly consisting of cleanup actions.

Message Handling

The most common part of exception handling consists of *message handling*. If a message needs to be sent, there are three things to be done: compose the content of the message (what); choose the destination of the message (to whom); and, choose the medium (how). Figure 5.3 shows these three steps:

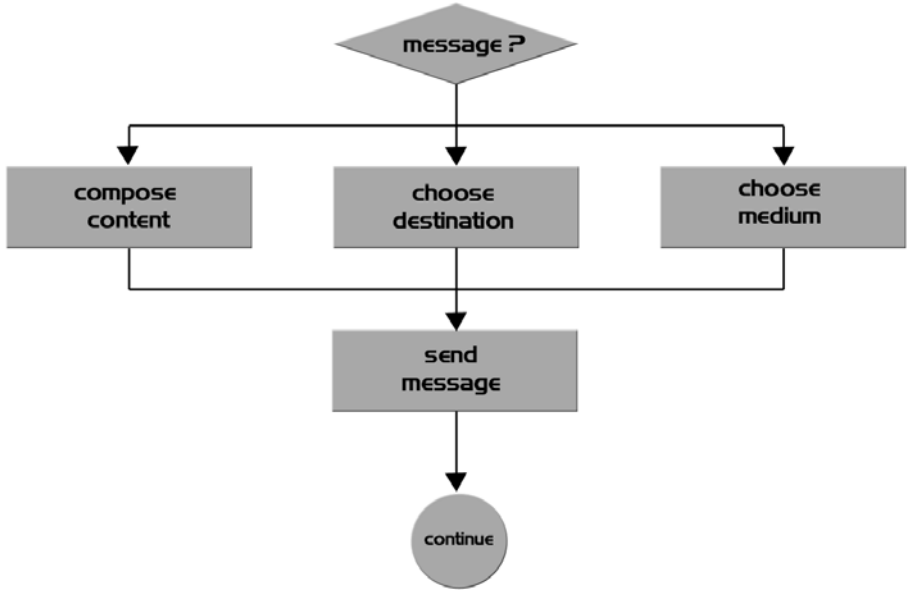


Figure 5.3 Message Handling Elements

Because most ABAP developers are already familiar with implementing regular message handling in ABAP (using the `MESSAGE` command), we won't elaborate further on that subject. However, we do want to emphasize an attractive ABAP feature—application logging—which is used less frequently than it warrants. This way of collecting and reporting exceptions is particularly useful in situations where the end user cannot directly be involved in the exception handling process. Therefore, programs running in the background will benefit most from this feature.

In principle, the same basic topics (what, to whom, and how) are as relevant for application logging as they are for message handling. With application logging, the destination of a message is defined in *objects* and *sub-objects* with Transaction SLG0 (see Figure 5.4).

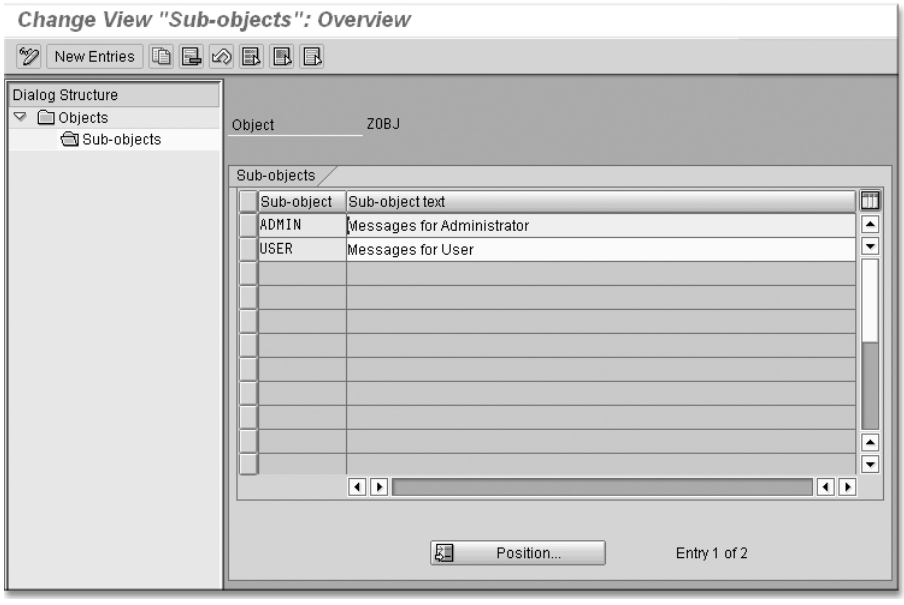


Figure 5.4 Customizing Application Log Objects and Sub-objects with Transaction SLGo

The first advantage of application logging is that its sub-objects enable you to send different types of exception information to the different people involved. For example, an administrator may benefit most from knowing the program name and the exact line number where an exception has occurred, whereas an end user may be interested primarily in knowing which data was actually skipped during the processing of a background job.

The second advantage of application logging is that it permits you to add both help information and free text to a message. Furthermore, messages can either be stored or displayed immediately. This means that the use of the Application Log is not limited to only background processing.

The third advantage of application logging is that existing functionality can be reused. Various standard function modules are available. Up to Release 4.6, this pertains to function modules whose names start with "appl…". Since Release 4.6, however, new function modules whose names start with "bal…" are also provided, but the aforementioned function modules are still supported.

Now, we'll briefly mention the latest versions of these function modules and their associated functionality. To create an exception object, use function module `bal_log_create`. The interface parameters to be supplied are identifying parameter values such as object, sub-object, and, possibly, your own extra identifying information (e.g., a document number). To add a message to the Application Log, you

use function module `bal_log_msg_cumulate`. To store the Application Log with all the messages contained in it in the database, use function module `bal_db_save`.

Finally, if the Application Log cannot be viewed immediately in your program, standard Transaction SLG1 is available for displaying it (see Figure 5.5).

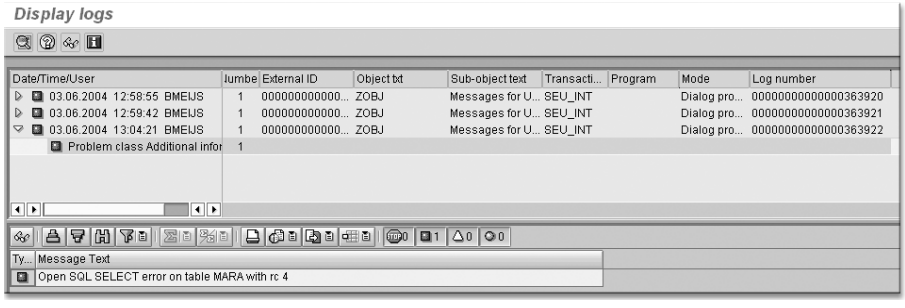


Figure 5.5 Display Application Log Information Using Transaction SLG1

Cleanup Actions

In addition to message-handling activities, the other important thing to consider is whether the program can continue, and if so, how. You may, for example, be able to skip an erroneous record in a file and continue as before with the rest of the data. But, even if you think you can continue, you first need to clean up the mess. In the first place, this means that you need to ensure that all the intermediate variables used still contain the correct values—this could require either initializing data (with a `CLEAR` or `REFRESH` command), or resetting data to the value it contained just before the exception occurred, for example, resetting counter information. See the following simple example of a `CLEAR` action:

```
SELECT SINGLE * FROM mara INTO wa_mara
              WHERE matnr = pa_matnr.
IF sy-subrc NE 0.
   CLEAR wa_matnr.
ENDIF.
```

If an update has failed, you should probably execute a `ROLLBACK WORK` command in order to restore the proper state of the database. Note that a database rollback is required regardless of whether the program can continue. Inconsistent database updates must always be reversed.

5.3 Class-Based Exception Handling

One of the basic dilemmas, already mentioned briefly in Section 5.1, is that adding appropriate exception handling to ABAP code can be a laborious task. This is one of the main reasons why adding exception handling code to customized ABAP programs is frequently avoided if it isn't absolutely necessary. In particular, there is one basic contradiction that complicates exception handling. On the one hand, actually executing the *handling* part of exception handling is not always appropriate in the exact place where the exception is raised. On the other hand, you want to have as much detailed context information available about an exception as possible. For example, a function module itself may notice an exception, but nevertheless have to let the calling program do the associated exception handling.

Letting the calling piece of code (or the piece of code calling that piece of code) do the exception handling means that it must also have all the available detailed exception information. Passing all this data along the Call Stack can be particularly cumbersome, especially if it contains several levels. This is where the advantages of class-based exception handling come into play. We'll discuss these advantages in the next sections. In Sections 5.3.1 and 5.3.2, we'll show you a basic implementation of class-based exception handling and discuss the most important characteristics of an exception class. In Section 5.3.3, we'll discuss the benefits of class-based exception handling. Next, in Section 5.3.4, we'll show you where to start if you want to add class-based exception handling on top of your current exception handling. Finally, in 5.3.5, we'll briefly explain how you can create your own exception classes.

5.3.1 Basic Implementation

We'll start with a basic implementation of (ABAP Objects) class-based exception handling in the next piece of sample code. The `RAISE EXCEPTION TYPE <exception class>` command is used to generate (raise) an exception. The `TRY-ENDTRY` block represents the area in which intercepting an exception is possible. The `CATCH` statement is used to actually intercept an exception and trigger exception handling. In this particular piece of code, two exceptions are taken into account: a divide-by-zero exception, and a SQL error:

```
REPORT /ctac/bk12exchandling_5a.
DATA: tp_average TYPE i,
      tp_total   TYPE i.

START-OF-SELECTION.
*
  TRY.
* This will raise cx_sy_sqlerror
```

```
      tp_average = tp_total / 0.           "SIGNAL implicit

      RAISE EXCEPTION TYPE cx_sy_sql_error."SIGNAL Explicit

    CATCH cx_sy_zerodivide.                "Intercept
* Handle exception here                    "ACT
    CATCH cx_sy_sql_error .                "Intercept
* Handle exception here                    "ACT
  ENDTRY.
```

5.3.2 Exception Classes

The fact that an exception is implemented as a class in ABAP Objects implies that it also has some of the main characteristics of a class: an exception class will always have one superclass and may have various subclasses. In fact, every exception class that is created must refer to one of three standard superclasses: CX_STATIC_CHECK, CX_DYNAMIC_CHECK, or CX_NO_CHECK. These standards all have class CX_ROOT as their superclass. Other exception classes, including those that you create yourself, must have a direct or indirect relation with one of the aforementioned three superclasses. An exception class tree may look like the one shown in Figure 5.6:

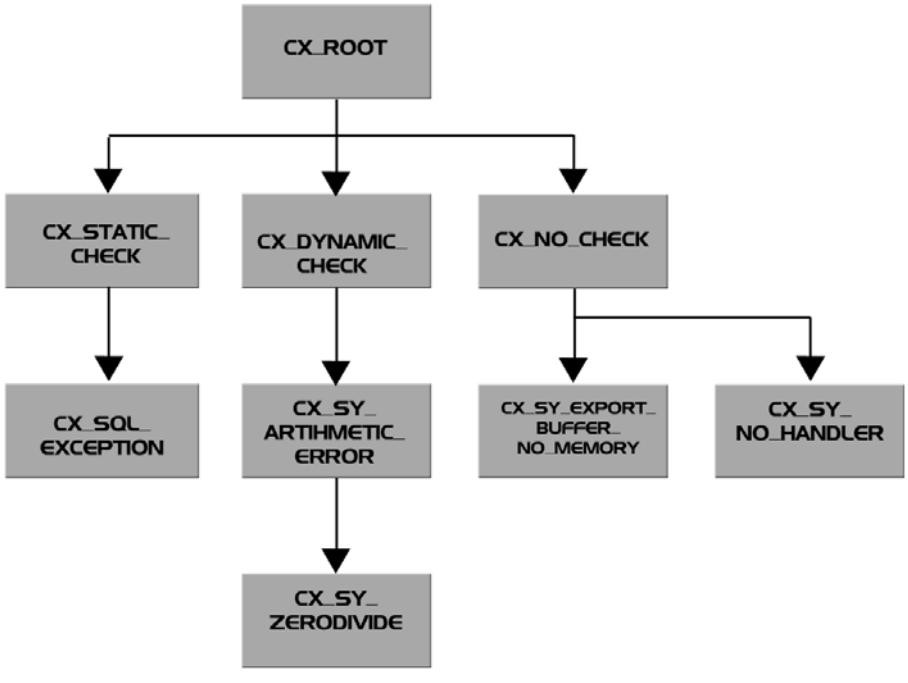


Figure 5.6 The Exception Class Tree

For example, according to Figure 5.6, class CX_SY_ZERODIVIDE inherits some of its attributes from CX_SY_ARITHMETIC_ERROR, which in turn inherits some of its attributes from CX_DYNAMIC_CHECK. It is not possible to inherit attributes directly from class CX_ROOT. In the next section, we'll describe the benefits of class-based exception handling.

5.3.3 Details of Class-Based Exception Handling

The use of class-based exception handling has two basic advantages:

- Passing on information about exceptions through the Call Stack is done by providing a reference to the exception object raised using a `RAISING` parameter. Note that (in contrast to traditional exception handling) the details of an exception don't have to be passed on because they're already contained in the exception object.
- Exception information can be made as context-specific as you want. If required, you can add your own context-information.

This means that you can *pack* all possible detailed information about an error into the exception object immediately at the moment when the exception occurs. You don't have to decide exactly what specific information needs to be passed on to a previous level in the Call Stack. Instead, you need only indicate in which exception object this information is stored.

How this works in real-time is explained most easily by showing various examples. We'll discuss the following: the timing of catching exceptions; the level at which interception takes place; where to intercept which type of exception; an implementation of context-specific exception information; and finally, an implementation of cleanup actions.

Timing the Interception of a Class-Based Exception

The basic advantages of class-based exception handling imply that intercepting an exception and executing the associated exception handling can be two independent actions. Hence, you can determine the best moment for actually intercepting an exception yourself. The following code example illustrates this principle:

```
REPORT /ctac/bk12exchandling_5.
* ...
START-OF-SELECTION.
*
  TRY.
      CALL FUNCTION '/CTAC/FM_WITH_CLASS_BASED_EXC2'
        EXPORTING
          itp_input  = tp_input_fm
```

```
        IMPORTING
          etp_export = tp_output_fm.

    CATCH cx_sy_zerodivide.       "Intercept
* Handle exception here           "Act
    CATCH /ctac/cx_dynamic_check. "Intercept
* Handle exception here           "Act
  ENDTRY.

FUNCTION /ctac/fm_with_class_based_exc2.
*"----------------------------------------------------------------------
*"*"Local interface:
*"  IMPORTING
*"     REFERENCE(ITP_INPUT) TYPE  CHAR10
*"  EXPORTING
*"     REFERENCE(ETP_EXPORT) TYPE  CHAR10
*"  RAISING
*"      CX_SY_ZERODIVIDE
*"      /CTAC/CX_DYNAMIC_CHECK
*"----------------------------------------------------------------------
  DATA: ltp_total   TYPE i VALUE 10,
        ltp_average TYPE i.
  TRY.
* This will lead to a zero divide error
      ltp_average = ltp_total / 0.                  "Signal implicit
*
      RAISE EXCEPTION TYPE /ctac/cx_static_check.   "Signal explicit
*
      RAISE EXCEPTION TYPE /ctac/cx_dynamic_check.  "Signal explicit

    CATCH /ctac/cx_static_check.                    "INTERCEPT
* Exception handling here                          "ACT
  ENDTRY.
ENDFUNCTION
```

The function module `/ctac/fm_with_class_based_exc2` that is called in the code considers (raises) three possible exceptions. The first exception is an implicitly raised runtime error (CX_SY_ZERODIVIDE). The other two exceptions (/CTAC/CX_STATIC_CHECK and /CTAC/CX_DYNAMIC_CHECK) are explicitly raised by a `RAISE EXCEPTION TYPE <exception>` command. Only one of the three exceptions mentioned (CX_STATIC_CHECK) is actually intercepted within the function module itself. The other two (CX_DYNAMIC_CHECK and CX_SY_ZERODIVIDE are intercepted one level up in the Call Stack by using the `CATCH` command. To enable the calling code to do this, the function module passes on reference information about the last two exceptions in `RAISING` parameters. Note that the function module loses its influence on the exception handling by passing on this information. If the calling code doesn't do anything with the provided reference information, a program dump is generated.

The Class Level of the Intercepted Exception

In the sample code above, where we discussed the timing of catching exceptions, each single exception passed along the Call Stack is also intercepted separately. However, class-based exception handling allows you to intercept exceptions in a more general way. Consider the following code:

```
REPORT /ctac/bk12exchandling_5a.
DATA: tp_average TYPE i,
      tp_total   TYPE i.

START-OF-SELECTION.
*
  TRY.
* This will raise cx_sy_sql_error
      tp_average = tp_total / 0.           "Signal (implicit)

      RAISE EXCEPTION TYPE cx_sy_sql_error."Signal (explicit)


    CATCH cx_sy_sql_error.                 "Intercept
* Handle exception here                    "Act
    CATCH cx_root.                         "Intercept (cx_sy_zerodivide)
* Handle exception here                    "Act
  ENDTRY.
```

In this code, two exception signals are possible: an implicit exception (divide-by-zero), and an explicitly raised exception (of class CX_SY_SQL_ERROR). Only the latter exception is intercepted with an explicit reference to the exception class in the corresponding `CATCH` statement (`CATCH cx_sy_sql_error`). The implicit exception (divide-by-zero), on the other hand, is not referred to; however, it will be intercepted by the `CATCH` statement that refers to exception class CX_ROOT (`CATCH cx_root`). Recall that CX_ROOT is the superclass for all class-based exceptions. Therefore, intercepting a CX_ROOT exception implies that all exceptions of subordinate types that are not considered elsewhere will also be caught. This means that you can use the interception of CX_ROOT exceptions as a kind of extra safety net (similar to using a `WHEN OTHERS` condition in a `CASE` statement).

However, there is one particular disadvantage to intercepting exceptions on the level of CX_ROOT, namely, you lose the context-specific attributes that would be available in more specialized exception objects. Therefore, this kind of exception handling is not appropriate for every kind of exception. In general, it is best suited for intercepting runtime errors because not all possible runtime errors can be anticipated in your program.

Where to Intercept Which Type of Exception

Each type of exception has its own appropriate place where it can best be intercepted and handled. The three most important subclasses (CX_STATIC_CHECK, CX_DYNAMIC_CHECK, and CX_NO_CHECK) represent the basic exception types. The specific type of exception determines what happens when an exception is raised within a procedure and *not* directly intercepted or passed in the `RAISING` part of the procedures interface parameters:

- CX_STATIC_CHECK
  Exceptions of this type must either be caught directly in the procedure where they are raised, or passed (propagated) along the Call Stack using the `RAISING` option. A syntax error will occur if your code doesn't do one of these two things. Exception classes of this type are most appropriate for relatively predictable exceptions, since they should be explicitly raised in a procedure; for example, the static exception CX_SQL_EXCEPTION can be raised after a `SELECT` statement.
- CX_DYNAMIC_CHECK
  Exceptions of this type don't necessarily have to be intercepted immediately or passed (propagated) along the Call Stack. The syntax check doesn't verify this, only the runtime environment does. If an (implicitly raised) exception of this type occurs in a procedure and is not caught there or propagated along the Call Stack, a new exception of class CX_SY_NO_HANDLER is raised by the system. Runtime errors, in particular, have this dynamic type. Because neither handling such exceptions directly in the procedure where they occur, nor propagating them, is practical, the best option is to catch these exceptions in one central place in your code. This repository is preferably located somewhere in the main program.
- CX_NO_CHECK
  Exceptions of this type cannot be passed along the Call Stack with the `RAISING` option. If an exception of this type occurs, the system will jump directly to the first relevant `CATCH` statement up the Call Stack and skip all other statements. Exceptions of this type are severe but highly unpredictable and are often related to system-wide problems, such as a problem with resources. Note that the aforementioned exception class CX_SY_NO_HANDLER is based on this type.

In SAP releases prior to 6.20, the propagating technique is not yet available. You can simulate it using regular interface parameters, however, the programming required is quite cumbersome. Therefore, we don't recommend applying this alternative in older releases.

Context-Specific Information About an Exception

The minimal information about a particular class-based exception that is always available is the name of the program causing the exception; the line where the exception occurred; the name of the exception; and the exception message that was generated. However, note that all kinds of additional attributes can be added on top. Consider the following piece of sample code to see how this works:

```
REPORT /ctac/bk12exchandling_5b.

DATA: rf_static    TYPE REF TO /ctac/cx_static_check,
      rf_root      TYPE REF TO cx_root.

DATA: tp_text      TYPE string,
      tp_extra_att TYPE char10,
      tp_program   TYPE syrepid,
      tp_linnr     TYPE i.

START-OF-SELECTION.

*
  TRY.
      PERFORM do_something.

    CATCH /ctac/cx_static_check INTO rf_static. "Intercept

      tp_text      = rf_static->get_text( ).     "Act
      tp_extra_att = rf_static->extra_attribute. "Act

      CALL METHOD rf_static->get_source_position "Act
        IMPORTING
          program_name = tp_program
          source_line  = tp_linnr.

  ENDTRY.
*
FORM do_something RAISING /ctac/cx_static_check.

  RAISE EXCEPTION TYPE /ctac/cx_static_check
                  EXPORTING extra_attribute = 'Extra Info'(001). "Signal

ENDFORM.                    "Do_something
```

When an exception of class /CTAC/CX_STATIC_CHECK is raised (marked bold at the end of the code) in `FORM do_something`, an extra attribute is also passed on in terms of an `EXPORTING` parameter (called `extra_attribute`). When the exception is intercepted, this extra parameter will automatically become available in the exception object that is instantiated (or created) by using the `INTO` addition

(see the CATCH statement marked bold). In this case, the exception handling triggered by the CATCH statement consists of, among other things, getting this extra attribute.

Let's also include a more extensive example. In the following code, class-based exception handling is combined with the use of application logging discussed earlier:

```
REPORT  /ctac/bk12exchandling_4a   .
* This local class contains the actual functions for the appl. log
INCLUDE /ctac/exchandling_new.[2]

CONSTANTS  : co_log_object TYPE balobj_d  VALUE 'ZOBJ',
             co_log_subobj TYPE balsubobj VALUE 'USER',
             co_error      TYPE symsgty   VALUE 'E'.
* Reference variable for the exception handler class
DATA:        rf_appl_log   TYPE REF TO lcl_exception_handler.
* Reference variables for exception classes
DATA:        rf_sql_error  TYPE REF TO /ctac/cx_sql_error,
             rf_log_error  TYPE REF TO /ctac/cx_log_error,
             rf_root       TYPE REF TO cx_root.
* Variables for the application log
DATA:        tp_nrext      TYPE balnrext.
* Context information
DATA:        tp_subrc      TYPE i,
             tp_text       TYPE string,
             tp_program    TYPE syrepid,
             tp_linno      TYPE i.

DATA:        wa_mara       TYPE mara.

PARAMETERS:  pa_matnr      TYPE matnr OBLIGATORY.

START-OF-SELECTION.

  TRY.

      SELECT SINGLE * FROM mara INTO wa_mara
                    WHERE matnr = pa_matnr.

      IF sy-subrc NE 0.
        tp_subrc = sy-subrc.
        RAISE EXCEPTION TYPE /ctac/cx_sql_error         "Signal
                        EXPORTING sqlcode = tp_subrc
                                  tablename = 'MARA'.
      ENDIF.
* Intercept this SQL error
    CATCH /ctac/cx_sql_error INTO rf_sql_error.         "Intercept
```

2 See the appendices for this include.

```
      TRY.
* Get the information out of the exception object
          tp_text      = rf_sql_error->get_text( ).     "Act

* Create the exception handling object
          tp_nrext = pa_matnr.
          CREATE OBJECT rf_appl_log
                 EXPORTING itp_object    = co_log_object
                           itp_subobject = co_log_subobj
                           itp_extnumber = tp_nrext.
* Add 1 message to the application log
          CALL METHOD rf_appl_log->free_message_to_appl_log
            EXPORTING
              itp_msgty = co_error
              itp_text  = tp_text.
* Store the messages in the database
          CALL METHOD lcl_exception_handler=>write_to_db.
* Intercepting errors during processing of the logfile
        CATCH /ctac/cx_log_error INTO rf_log_error.
* Error handling here
        CATCH cx_root            INTO rf_root.
* Error handling here
      ENDTRY.
    CATCH cx_root            INTO rf_root.
* Error handling here
  ENDTRY.
```

In this code, when selecting data from database table MARA, a SQL error is detected by checking the associated returncode (`sy-subrc`). An explicit exception is then raised, intercepted, and retrieved from the exception object, and finally stored in the Application Log. If exceptions occur, they will become visible in the Application Log (by using Transaction SLG1).

Executing Cleanup Actions

As discussed earlier, the actual exception handling consists of more than just collecting messages: all kinds of *cleanup actions* must also be performed such as initializing variables and performing a rollback in the event of erroneous updates. In class-based exception handling, the cleanup actions that are also part of this process are explicitly separated from the rest of exception handling. The following code shows how this is implemented:

```
DATA: ltp_subrc type i.
TRY.
    wa_luw_1-key1 = itp_key1.
    wa_luw_1-notkey = itp_notkey.
    INSERT /ctac/tbk_luw_1 FROM wa_luw_1.
```

```
    IF sy-subrc NE 0.
      ltp_subrc = sy-subrc.
      RAISE EXCEPTION TYPE cx_sy_sql_error         "Signal
                      EXPORTING sqlcode = ltp_subrc.
    ENDIF.

    wa_luw_2-key1 = itp_key1.
    wa_luw_2-posnr = itp_posnr.
    wa_luw_2-somedata = itp_somedata.
    INSERT /ctac/tbk_luw_2 FROM wa_luw_2.

    IF sy-subrc NE 0.
      ltp_subrc = sy-subrc.
      RAISE EXCEPTION TYPE cx_sy_sql_error         "Signal
                      EXPORTING sqlcode = ltp_subrc.
    ENDIF.

  CLEANUP.
* The exception is not intercepted, but a rollback is really necessary
    ROLLBACK WORK.
ENDTRY.
```

The code contains INSERT actions on two different database tables. If one of the two INSERTS is not successful, the other database INSERT must be reversed as well. Note that the code does not contain any CATCH statement to intercept an error, whereas it does have a separate CLEANUP section for executing the ROLLBACK WORK. This means that a rollback will always be executed if there is an error in one of the database inserts, even if the error is not caught immediately within the TRY-ENDTRY block.

5.3.4 Making Existing Exception Handling Class-Based

Thus far, we have emphasized how to apply class-based exception handling as such, but we haven't explicitly mentioned that it is fairly easy to make existing exception handling class-based. All you need to do is to convert a *traditional* exception signal into a class-based signal (see Figure 5.7).

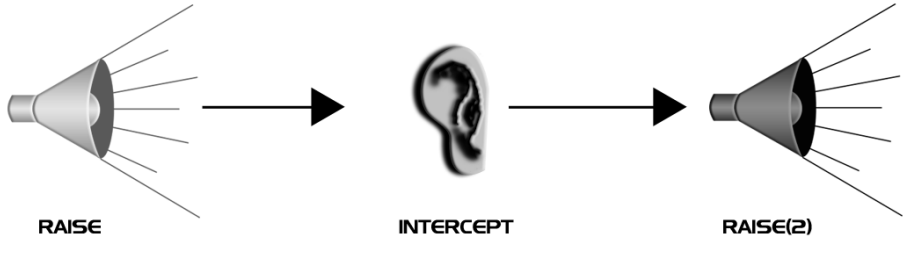


Figure 5.7 Intercept an Exception and Raise Another One

Converting a traditional signal such as a returncode into a class-based signal is done by inserting the `RAISE EXCEPTION TYPE <exception>` command at the place where otherwise traditional exception handling would be triggered. That's all you need to do. From then on, you must do things in a class-based way. The following code shows how a traditional signal is converted into a class-based signal:

```
TRY.
 CALL FUNCTION '/CTAC/FM_WITH_TRAD_EXCEPTIONS'
   EXPORTING
     itp_input        = tp_input_fm
   IMPORTING
     etp_export       = tp_output_fm
   EXCEPTIONS
     not_found        = 1
     OTHERS           = 9.

 CASE sy-subrc.
   WHEN 0.
* Everything is OK.
   WHEN 1.
    RAISE EXCEPTION TYPE /ctac/cx_notfound.
   WHEN 9.
    RAISE EXCEPTION TYPE /ctac/cx_others.
 ENDCASE.

  CATCH /ctac/cx_notfound.
* Action needed here..
  CATCH /ctac/cx_others.
* Action needed here..
ENDTRY.
```

As you can see, instead of doing the follow-up of the exception immediately after the `WHEN` conditions are checked, you can raise another kind of exception.

5.3.5 Creating Your Own Exception Classes

In the SAP IDES system Release 6.20, which we used for all the examples of ABAP code and the screenshots, we found 1002 global exception classes.[3] So, you should be able to find a global exception class that best suits your needs. If necessary, you can always create your own exception classes. This enables you to collect your own exception attributes and documentation.

3 Exception classes all begin with the prefix 'CX_'. The allowed customer names are 'ZCX_', `YCX_', or '/namespace/CX_'.

Exception classes are global classes. They must be created with the Class Builder (Transaction SE24, see Figure 5.8). They inherit the methods `get_text`, `get_longtext`, and `get_source_position` from the exception class CX_ROOT.

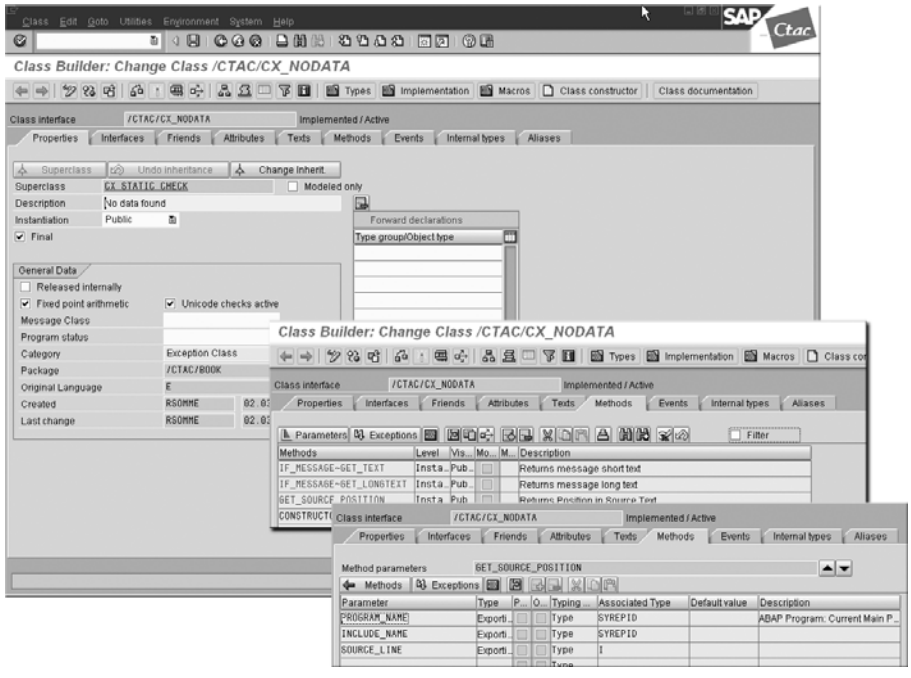


Figure 5.8 Definition of Exception Class /ctac/cx_nodata

In Figure 5.8, you can see how exception class /CTAC/CX_NODATA is defined as a subclass of CX_STATIC_CHECK, which, in turn, is a subclass of CX_ROOT. The methods `if_message~get_text` and `if_message~get_longtext` are defined in interface `if_message` and implemented in exception class CX_ROOT. These methods provide the error message text that is stored in the exception object. The method `get_source_position` is defined and implemented in CX_ROOT; it supplies information on the name of the program, the include, and the source line where an exception is raised.

Creating your own exception classes is appropriate when developing your own functionality with the ABAP Development Workbench. Exceptions that are specific for this new functionality can be raised within your ABAP code, supplying all the relevant context information.

5.4 Conclusions

In this chapter, we discussed the most important aspects of exception handling. We can summarize these aspects in the form of the following guidelines. First, we recommend that you *always* intercept every possible exception and take the appropriate action. Not knowing how to handle an exception is no excuse: if you don't expect one to occur, you can always stop the program.

Secondly, we propose that you consider application logging for message handling because of the standard features it offers.

Third, above all, we recommend implementing class-based exception handling as soon as you have access to the SAP Web Application Server Release 6.20. This offers more flexibility and reusability than traditional types of exception handling, particularly for more complicated situations in which exception information must be passed on through the Call Stack. Raising, intercepting, and actually handling exceptions are strictly separate activities. Cleanup actions are fully separated from other exception handling activities. It's relatively simple to detect exceptions in all your procedures by adding a `TRY` (at the beginning) and an `ENDTRY` command (at the end). Unforeseen exceptions can always be intercepted in the mainline of your ABAP programs for exceptions of class CX_ROOT and CX_SY_NO_HANDLER. In short, we strongly urge you to take advantage of all these opportunities.

Index

E

F

G

H

T

U

V

W

X