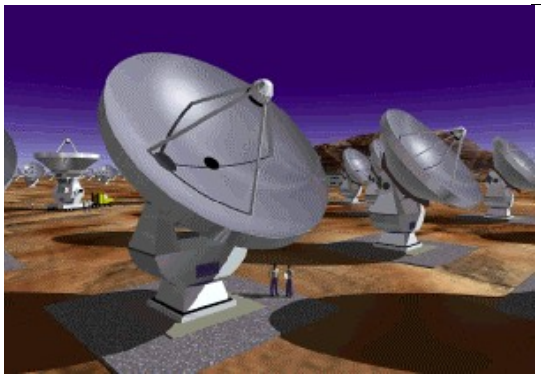


Atacama Large Millimeter	ALMA-SW-NNNN Revision: 4.1 2007-03-30

ACS Error System

Software Architecture and How-to manual

Bogdan Jeram (bjeram@eso.org),
Gianluca Chiozzi (gchiozzi@eso.org)
ESO
Mark Plesko (mark.plesko@ijs.si)
Jozef Stefan Institute
David Fugate (dfugate@ucalgary.ca)
University of Calgary
Sohaila Roberts (*sroberts@nrao.edu*)

Keywords:	
Author Signature:	Date:
Approved by: Institute:	Signature: Date:
Released by: Institute:	Signature: Date:

Change Record

REVISION	DATE	AUTHOR	SECTIONS/PAGES AFFECTED
	REMARKS		
1.0	2002-10-29	Bogdan Jeram	All
	Created		
2.0	2003-12-15	Bogdan Jeram	All
	Updated to the new error system Added C++ implementation Added XML schema definition		
3.1	2004-05-14	Bogdan Jeram	C++
	Update to ACS 3.1 implementation		
4.1	2005-06-02	Bogdan Jeram	All
	Updated to ACS 4.1 implementation. Added description of completion/error members. Improved examples.		
	2007-03-30	Nicolas Barriga	All
	Added information on how to use Completions as out parameters.		
	2007-05-02		
	Made some clarification about sending completion as output parameter,		

REVISION	DATE	AUTHOR	SECTIONS/PAGES AFFECTED

Table of Contents

1 Introduction

1.1 Scope

This document describes the architecture and design of the ACS Error System and explains how to use it. There are sections describing the error system in general and a section for each implementation: C++, Java and Python.

1.2 Glossary

http://www.alma.nrao.edu/development/computing/docs/joint/draft/Glossary.htm

XSLT eXtensible Stylesheet Language Transformations

1.3 References

[1] ALMA reviewed document 0005, **ALMA Common Software Technical Requirements**, G. Raffi, B. Glendenning, 2000-JUNE-05.

[2] ALMA reviewed document 0016, **ALMA Common Software Architecture**, G.Chiozzi, B.Gustafsson, B.Jeram, 2001-SEPT-09.

[3] Error System on-line documentation: $ACSROOT/man/api/html/acserr_8h.html

[4] OCI, TAO Developers Guide version 1.1a Chapter x: Error Handling

[5] sd&m, **ARCUS Error Handling for Business Information System**, Klaus Renzel

[6] **ACS Basic Control Interface**, M. Plesko, G. Tkacik, G.Chiozzi

[7] **Ten Rules for Handling Exception Handling Successfully,** Harald M. Müller

[8] **Ten Guidelines for Exception Specification**, Jack W. Reeves, http://agni.csa.iisc.ernet.in/ProgrammingLanguages/C++/cppr9607.c.reeves.html

[9] **More Effective C++**, S. Meyers Chapter: Exceptions

[10] **The C++ Programming Language**, B. Stroustrup, Chapter: Exception Handling.

1 General

Error systems are an essential and important part of almost every software product, and ACS is no exception. ACS is a framework for developing distributed and heterogeneous

software (in terms of programming languages, operating systems, communications, etc.) and the ACS Error System has to provide functionality for efficient error handling.

The ARCUS Error Handling for Business Information Systems 1.3 document provides a very good introduction to error handling concepts and introduces a complete pattern language for object-oriented design of an error system. The ACS Error System matches very well with what is described in the document, and we will often make reference to it.

The ACS Error System provides support for error handling in ACS itself as well as error handling in software based on ACS – all its features are available to developers.

An introduction to error handling terminology is provided in section 1.2 of 1.3.

The ACS Error System not only provides the means to handle fault situations, i.e. to detect and report errors, but it is more general. It also can be used to report how an action completed (in successful cases). An action can complete in one of two ways: with an error or successfully. An action can *fail* in many different ways and it can also *succeed* in several different ways which can also be reported using the ACS Error System. In the first case, we are talking about **errors,** while in both cases the term **completion** (i.e., **errors** are just a subset of **completions**) applies. There are some differences between the two cases, for example the way in which they are propagated (e.g. errors can be propagated using an exception mechanism) and there are different structures defined for keeping error or completion information. Details are covered in the subsequent chapters.

1.1 Requirements and goals

The error system for ACS must fulfill requirements from the “ALMA Common Software Technical Requirements” document 1.3 in addition to internal ACS requirements, such as:

1. provide support for a distributed and heterogeneous system
2. support error handling in synchronous and asynchronous cases
3. make it easy to define new error conditions in the system
4. support different programming languages, (e.g. C++, Java and Python)
5. easy to use API for all programming languages supported by ACS

The main goals of the error system are:

- defining errors/completions – how errors/completions are represented and organized
- defining structures to exchange error/completion information
- defining how error information is propagated both locally and remotely (error/completion reporting mechanism)

2 Architecture and Design

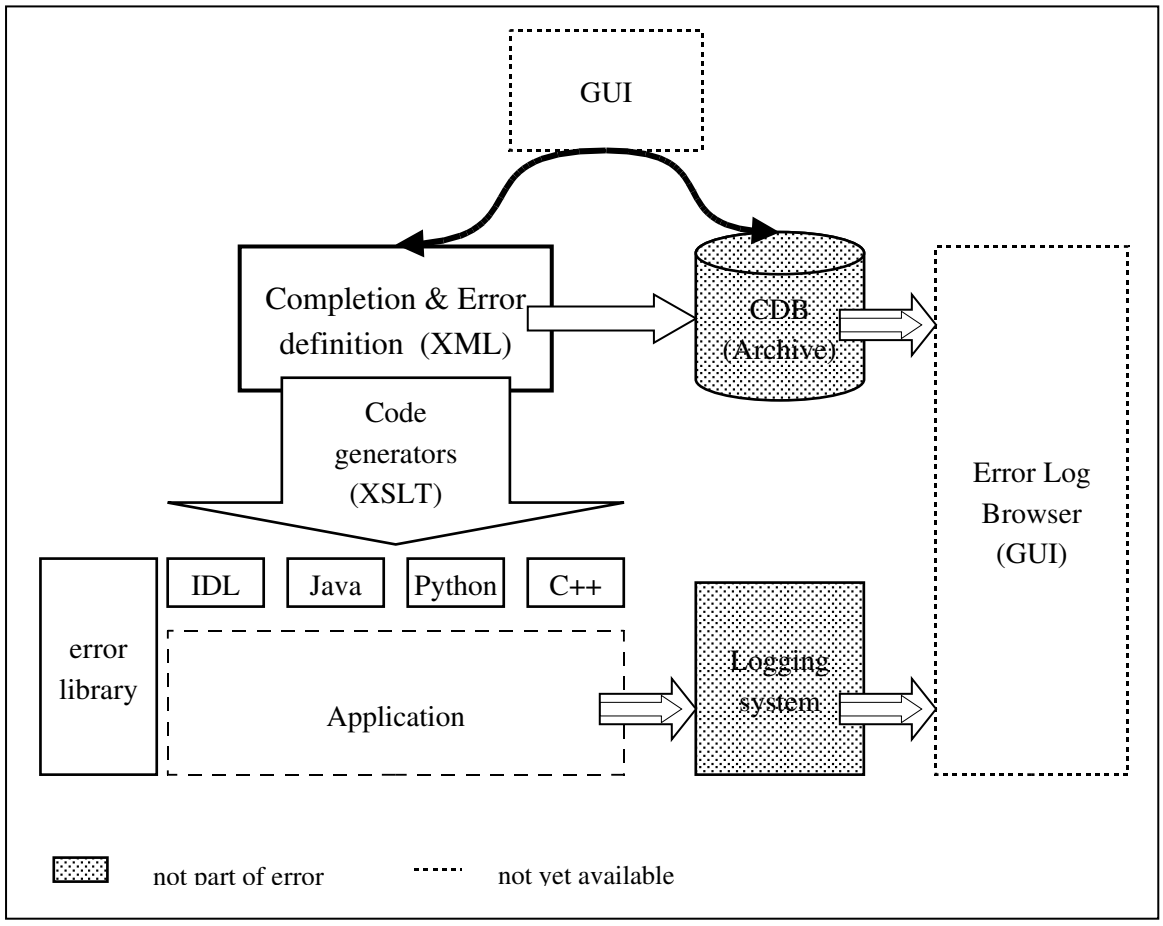


Figure 1 Architecture of the error system

The figure above depicts the architecture of the ACS Error System, which at the most basic level consists of:

- completion and error definitions

Errors and completions are introduced into the system using XML files. XML definitions are used to define completion and error codes and put run-time information into the configuration database about completions and errors (e.g., descriptions, etc.).

The ACS Error System defines an XSD schema to describe completions and errors. Currently XML definition files based on this schema are written by hand or with off the shelf XML editors.

TBD: In the future we will implement a custom GUI for editing error information. Such a GUI will be responsible for putting the error/completion information into the configuration database.

- **code generation**

The error system utilizes XSLT code generation. From each definition, source code is generated. Code is generated for IDL as well as the supported programming languages (i.e., C++, Java, and Python) using distinct XSLT transformations. Applications manipulate the error system primarily through this generated code, which in turn is dependent upon the error library.

- **common code (error library)**:
The error library contains common functionality for all generated code, such as logging, memory management, etc. There is a library for each programming language.

- **GUI log error browser**

Applications which use the error system can log completions/errors into the ACS Logging System (**Centralized Error Logging pattern in 1.3**). From the logging system, error logs can be retrieved using the generic GUI logging client (i.e., jlog).
TBD: In the future, a GUI application developed solely for browsing error logs will be created. This GUI will allow browsing, filtering, listing, etc. of error logs obtained from the error system via the logging system.

2.1 Dependency

The ACS Error System is dependent on two subsystems: time and logging. It needs the time subsystem to obtain information for the error's timestamp. The logging subsystem is used to log error (trace) information into the Centralized Logger (if one is available).

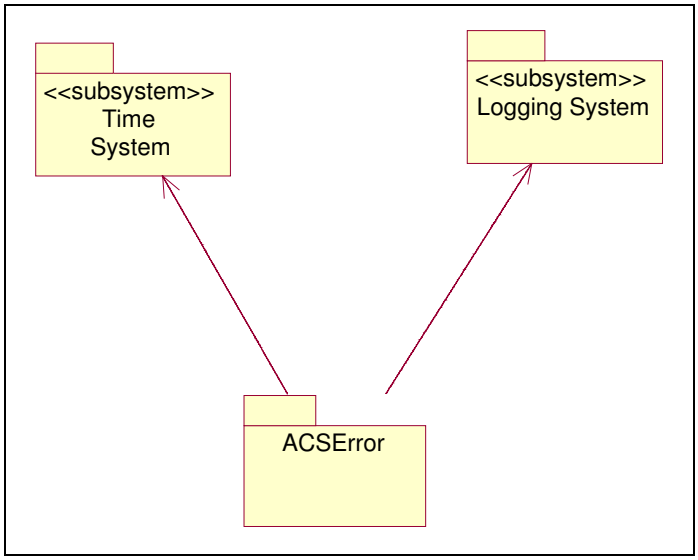


Figure 2: Dependency diagram

2.2 Completion/error definition and organization

2.2.1 Representation

Each error/completion is represented by two identifiers:

- **type**
- **code**

Types are used to organize related errors/completions into groups. Types and codes are hidden from the end user. From an object-oriented perspective, they are mapped into namespaces (modules in IDL) and classes as follows:

- **type** is mapped into a **namespace** (**IDL module**), i.e. for each type a new namespace (IDL module) is defined
- **code** is mapped into a **class** inside the "type" namespace

Then, a user simply instantiates an (error) object of the correct type which represents an error/completion code, following the **Error Object pattern 1.3**.

2.2.2 Description

Each type and code contains a **short** and **long description** of the error/completion. Furthermore it can contain a **URL** where more information about the error can be obtained.

All of this information: types, codes, and descriptions must be defined by the person who creates the error/completion for use in the ACS Error System. In the case of errors/completions provided by ACS, ACS provides the definitions. This information is provided in XML format.

2.2.3 Error context

Each error is accompanied by additional information which helps to reconstruct where, when, and how the error occurred. The following error context information can be obtained at compile or at run-time by the error system itself or by the developer:

- **source code error information** which can be obtained at compile time: filename, line number, and routine name. Some or all of this information is obtained automatically by the error system. However, the user is always free to set/change any of them.
- **run-time error information** is gathered at execution time: host name, process name, and thread id. Under normal circumstances, all run-time information is obtained automatically by the error system.
- **timestamp** tracks the epoch when the error or completion occurred.

NOTE: timestamp is provided for both error and completions, while the other information is provided solely in the context of an error.

2.2.4 Severity

For each error a **severity** level can be specified: `Error` (default), `Critical, Alert,` or `Emergency`. So be sure to look for log messages with all of these priorities when you are looking for error log messages.

2.2.5 Arbitrary data and completion/error members

There is also the possibility of adding arbitrary user-defined information to a completion/ error. This is used to tailor an error object to the current context.

There is a generic way to add (set) and retrieve (get) arbitrary user-defined information at run-time in **name-value** (CORBA string, CORBA string) form. For such user-defined information, the name and the type of the data are normally not known in advance (i.e. at the time of error/completion definition); as a consequence, the data type cannot be

checked at compile time. In addition, for this type of user-defined data there is no information describing the arbitrary data (i.e. metadata).

To solve the problems with user-defined name-value pairs described above, completion/error **members** were introduced. Members are used in a completion/error definition to describe the name and the type of the arbitrary user-defined information at definition time. The values of these members are then get/set at run-time in a type safe manner through generated getter and setter methods. Since members are described in a generic way, all member data can be accessed (set/get) in a generic way, as well. Members also allow the *documentation* of arbitrary data accompanying the completion/error (i.e. the definition of metadata). In other words, when members are used, the user can determine by looking at the completion/error definition what arbitrary information can be get/set for a particular type of completion/error.

TBD: Members are currently only supported in C++. Support for Java and Python will be added in the future.

Example: instead of defining a new error for each file that cannot be read, we define just one error: `FileNotFound` with a member `FileName` of type string. By setting the value of this member (=file name) at run-time in each instance, we are able to convey the particular file that was not found. This could also be done in a generic way by adding name-value pairs at run-time.

2.3 Completion/error handling structures

In the ACS Error System, all completions and errors that occur are encapsulated in objects – completion and error objects respectively. The ACS Error System provides a means to chain consecutive error conditions into a linked list of error objects within a single process. The linked list of error objects is called an error backtrace; see the **Backtrace pattern** (reference 1.3, section 2.4).

NOTE: error backtrace is used solely for reporting erroneous conditions, not successful ones.

2.3.1 Error backtrace

Error information is propagated using the concept of an error backtrace. The error backtrace allows tracing an error condition: through the cascade of calls from its occurrence, through the series of (distributed) processes written in possibly different programming languages, to the originator of the action – i.e. the end-user or application.

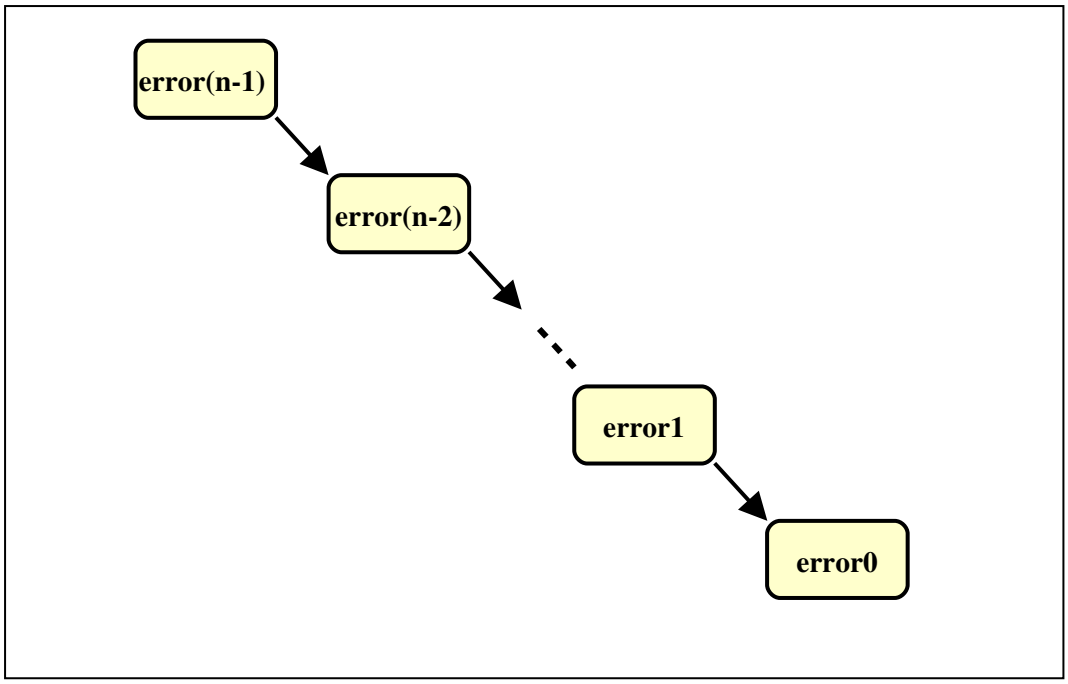


Figure 3 Error Backtrace

An error backtrace is realized as a linked list of error objects that is propagated through the chain of calls. At each level, information can be added – by adding a new item to the error backtrace and sending it up to the caller. Also, errors encountered at any level may be caught and recovered, resulting in the error backtrace being deleted.

In a layered architecture, erroneous conditions propagate across different layers. Whenever a layer cannot handle an erroneous condition, it passes it on to the next higher level. Because different layers present different levels of abstraction, each layer may only generate error reports at its own abstraction level, and each level may contribute to the buildup of the backtrace by adding a new error condition with additional context information (see **Exception Abstraction pattern in 1.3**).

The ACS Error System allows the user to build an error backtrace in a chain of calls. On each call level, the backtrace can be filled with context specific error information. An error backtrace can be built inside a single process and/or among distributed processes. Using the exception mechanism provided by modern object-oriented programming languages, it is possible to build an error backtrace; however its usage is limited due to the fact that exceptions cannot be used for asynchronous error reporting. Exceptions, of course, can be used wherever this limitation is not a problem.

Some languages (like Java) provide native support for automatically building an execution backtrace, but others (like C++) do not. However, in a distributed and multi-language environment, such as the ALMA project, it is not possible to rely on the backtrace features provided by a specific implementation language. Therefore, ACS has implemented an error backtrace structure that can be propagated through different processes, computers, and implementation languages - using CORBA as the transport mechanism. For inter-process communications such as CORBA method calls or

monitors, the ACS Error System serializes the error backtrace and transfers it to the calling process. There, it is de-serialized again into an error backtrace object.

An error backtrace cannot exist on its own; it can only exist as part of a **completion** or an **exception** structure (see 1.3 section 2.2).

2.3.2 Completion

The completion structure represents the status (i.e. completion state or result) of an operation (e.g., an action, event, value retrieval, etc.). Completion structures contain type, code, timestamp, and an error backtrace *when applicable* (i.e. if an error is being reported).

Depending on whether or not they contain an error backtrace, there are two types of completions defined:

- **error** completions are used for reporting failures, and thus contain an error backtrace. There is one generated class for each error code.
- **error-free** completions do not represent errors, and they **cannot** contain an error backtrace. Used to notify how a successful action was completed (i.e., finished). There is one generated class for each completion code.

In general, we can say that completions can be used where exceptions cannot:

- in asynchronous actions/communications. Usually, in these cases completion structures are returned using a callback mechanism.

- for reporting different success statuses of the operation

An interesting discussion about the usage of completions and exceptions can be found in section C.2 of 1.3.

2.3.3 Exceptions

Exceptions are thrown when an abnormal or error situation occurs (1.2.5 of 1.3). There is one generated exception class per error code. *Every* exception includes an error backtrace.

2.4 Propagation

In a distributed and concurrent environment, an error condition is propagated over a series of processes until it reaches the final client or application, which either fixes the problem or logs it (C.3 of 1.3). The error and completion reporting procedure must be the same regardless of whether the communication is synchronous, asynchronous, monitor, or event-based.

The error system propagates error messages through ACS, making use of OO technology (e.g., **exceptions** and serialization) where applicable. In cases where this is not possible (see Section 2.3.2), **completion objects** should be used. To summarize, completions and errors can be propagated using one of the following techniques:

- By throwing an **exception;** only error and abnormal conditions can be reported.
- By returning a **completion** which can be used for propagating both completions and errors.

Because ACS is used to build heterogeneous distributed systems, the error system must provide a way to propagate completion and error messages, not just **intra-process** (i.e., locally), but also **inter-process** (i.e., remotely). The representation of an error in both cases is the same. There are just minor differences between local and remote usage of the error system. In the remote case, error information is sent using mechanisms and constructs of CORBA distributed objects. In both cases we can use either exceptions or completions, as appropriate.

Additionally, the ACS Error System supports both **synchronous** and **asynchronous** calls. In the first case, it is necessary to wait until a (blocking) synchronous call completes to get the completion status and eventually error information. For asynchronous calls, however, completion information is sent to the originator of the call at the end of the asynchronous call's execution using a callback mechanism: a completion object (i.e., structure) is passed as one parameter of the callback method. In synchronous cases, exceptions are the preferred choice. For asynchronous cases, completions must be used (see section C.2 of 1.3).

In multithreaded environments, completions and errors can be propagated among threads using the **Multithreading Exception Handling pattern** (see 1.3, section 2.7 for details). ACS provides a C++ implementation of the **Exception Manager** class described in this design pattern.

2.5 An Example

In order to better understand how the ACS Error System works, consider the following example:

1. A GUI client issues a request to the telescope server for the telescope to point at a named star. Because this action takes a relatively long time, a remote asynchronous call is used. Information about action completion is provided via the callback mechanism and a completion structure.
2. The telescope server calls the star server to retrieve the position of the star. This call is done using a remote synchronous call (i.e. a method call on a remote object) and an eventual error is reported using the exception mechanism.

3. The star server calls the database to retrieve the star's coordinates using a normal local synchronous call (i.e. a normal method call).

4. The database server queries the database.

Let's assume that the query in the database server fails because the record cannot be found. As a consequence, each of the processes above encounters at least one error. The GUI client must get a completion which contains information about all of the problems (i.e., the error backtrace), including the true cause (e.g. "record not found"). To achieve this, at each level the error backtrace from the previous level has to be added to the newly created one.

Here is a scenario describing how an error is propagated in the above example:

1. The query error of the database server is propagated to the star server by creating and throwing a regular (i.e., non-CORBA) `RecordNotFound` exception.

2. The star server catches the `RecordNotFound` exception and creates a new CORBA exception, `PositionNotObtained,` and adds the error backtrace from the caught (`RecordNotFound)` exception.The new CORBA exception is then thrown remotely.

3. The telescope server catches the remote (i.e., CORBA) exception, `PositionNotFound.` It then creates a completion, `PointingFailure,` adding to it the previous error backtrace retrieved from the caught exception. The completion is then passed to the GUI client as one parameter of the method of its callback object. In the case of a non-failure (i.e., successful) scenario, the telescope server sends, at periodic time intervals, the Completion `PointingInProgress,` so that the GUI client is aware that pointing is still going on (i.e., the server is still alive).

At each level, additional (context specific) information can be added (see section 2.2.5), like the name of the record which was not found, the name of the star for which the position could not be obtained etc.

The implementation details of this example are discussed in the next section.

2.6 XML completion/error definition

For each type, one has to create a new XML file with type and (error) code definitions. This XML document must conform to the schema `ACSError.xsd` (see 8.8.1 for details). There, the `Type` XML element is defined with two required attributes:

- `name:` for specifying the name of the type (mnemonic)

- `type:` for specifying the type number, which must be unique in the system (and, therefore, there shall be a "central authority" responsible for allocating Type numbers; see section 4).

and a few optional attributes:

- `shortDescription:` a short description of the type can be provided
- `description:` a full description of the type
- `URL:` URL where more information about the type can be obtained
- `_prefix:` IDL prefix (default is alma)

The Type element contains 0 or more XML elements of the following types:

- `Code:` element where non-erroneous completions can be defined
- `ErrorCode:` element for erroneous completions and exceptions can be defined

All `Code` elements *must* be specified before `ErrorCode` elements, per the schema definition.

Both elements (`Code` and `ErrorCode`) must define the following (required) attributes:

- `name:` specifies the code name (mnemonic)
- `shortDescription:` short description of the code
- `description:` a full description of the code

and an optional attribute:

- `URL:` URL where more information about the code can be obtained

The `ErrorCode` element can additionally specify an optional attribute:

- `_suppressExceptionGeneration:` flag which indicates whether code for exceptions will be generated. The default value for this flag is false, which means that by default code for the exception is generated.

The `Code` or `ErrorCode` elements can specify 0 or more XML `Member` elements. To introduce a new member, the following (required) attributes must be defined inside the `Member` element:

- `name`: specifies the member name (name of arbitrary user-defined information)
- `type`: type of the member. Currently available types are: *`long`*, *`double`*, and *`string`*.
- *`description`*: description (i.e., documentation or metadata) for the memeber

Once the XML file with type/code definition is available, the developer has to configure the Makefile, as described in Section 3, to generate code.

Here is a sample XML file for the error type ‘ExmplErrType’ which contains three error codes and one non-error. For the last error (ErrorWOException), exception code is not generated because *_suppressExceptionGeneration* is set to true. The error *PointingFailure* contains two members of type double, *Azimuth* and *Elevation*.

```
1: <Type xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:noNamespaceSchemaLocation="ACSError.xsd"
        name="ExmplErrType"
        type="900902"
        _prefix="alma"
   >
2: <Code name="PointingInProgress"
         shortDescription="Pointing in progress"
         description="Pointing in progress"
   />
3: <ErrorCode name="PointingFailure"
              shortDescription="Pointing Failure"
              description="Pointing cannot be executed"
   >
   <Member name="Azimuth"
           type="double"
           description="Azimuth"/>
   <Member name="Elevation"
           type="double"
           description="Elevation"/>
   </ErrorCode>
4:
5: <ErrorCode name="PositionNotObtained"
              shortDescription="Position not obtained"
              description="Position of the object cannot be
              obtained"
   />
6: <ErrorCode name="RecordNotFound"
              shortDescription="Record Not Found"
              description="Record cannot be found">
   <Member name="RecordName"
           type="string"
           description="Record name"/>
   </ErrorCode>
7: <ErrorCode name="ErrorWOException"
              shortDescription="Error w/o exception"
              description="An example of error w/o exception"
              _suppressExceptionGeneration=”true”
   />
8: </Type>
```

2.7 Code generation

The ACS Error System is based on XML-XSLT code generation.

Types and their corresponding codes are defined via XML files in a language independent way. For each type, a unique XML file must be defined by the developer. This file is used by the XSLT transformation to produce the code. Generated code contains a set of completion and exception helper classes – one for each error code.

There is a separate XSLT for each programming language and one specifically for IDL. All code generation is done automatically by using a target in the ACS Makefile.

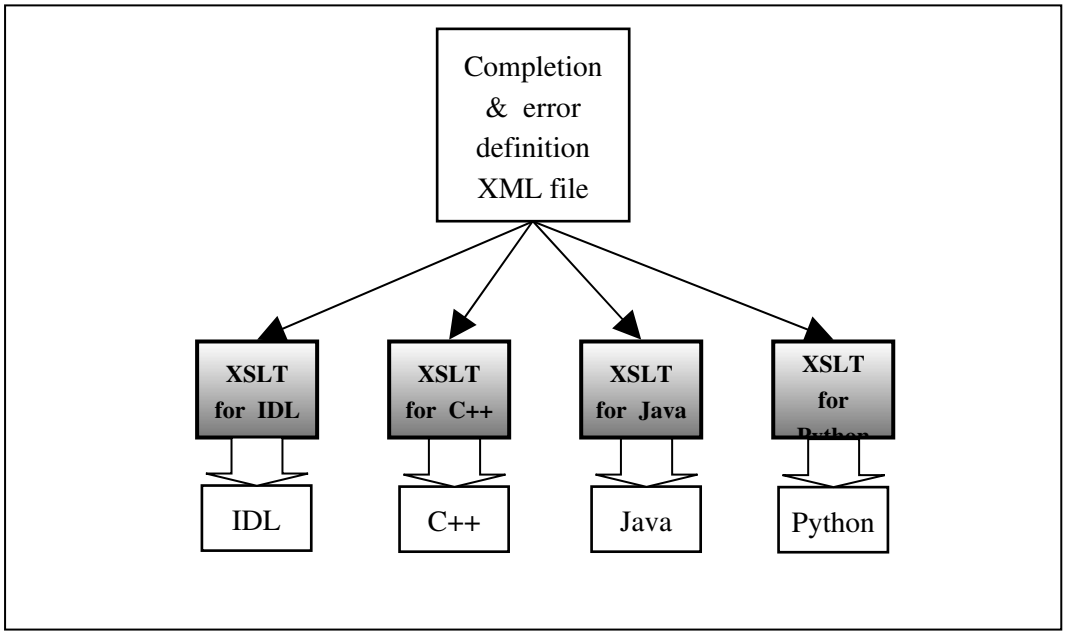


Figure 4 Error System code generation

3 Makefile Support

Code generation for IDL and all ACS-supported programming languages is done automatically with the support of the ACS Makefile. For this purpose the ACSERRDEF tag was introduced, where XML files with error definitions have to be specified. Here is an excerpt from a Makefile for generating code for ExmplErrType.xml (see 2.6) found in the "idl" directory:

```
…

ACSERRDEF = ExmplErrType

…
```

Besides code generation, the ACS Makefile takes care of proper installation of the generated files (i.e., “make install”) and their deletion (i.e., “make clean”).

4 Adding new types to the system

Adding a new error type to the ALMA system requires the following steps:

- obtain a unique type number. This work has to be coordinated with the Software Engineering team which ensures that you get a number that is unique system-wide. Error type can be allocated from a wiki page.
  For more details see 8.8.1.

- create a new XML file as described in section 2.6 and adhering to the schema (see Section 2.6 and 8.8.1), and put it (the XML file) in the `idl` directory of your module

- specify the XML file’s name (without extension) in the ACSERRDEF tag in the Makefile (see section 3)

A new (error) code can be added to a *pre-existing* type simply by adding a new XML element of the correct type (`Code` or `ErrorCode`) to the `Type` element in the XML file.

NOTE: before you add a new completion (`Code`) or error code (`ErrorCode`) into your (sub)system, consider the following:

- check if the same or an equivalent code already exists and can be (re)used
- check if the newly added code/type can be used somewhere else (i.e. its code is not (sub)system specific). In this case, the code should be added to a common location, for example within ACS.

5 IDL

5.1 IDL common code

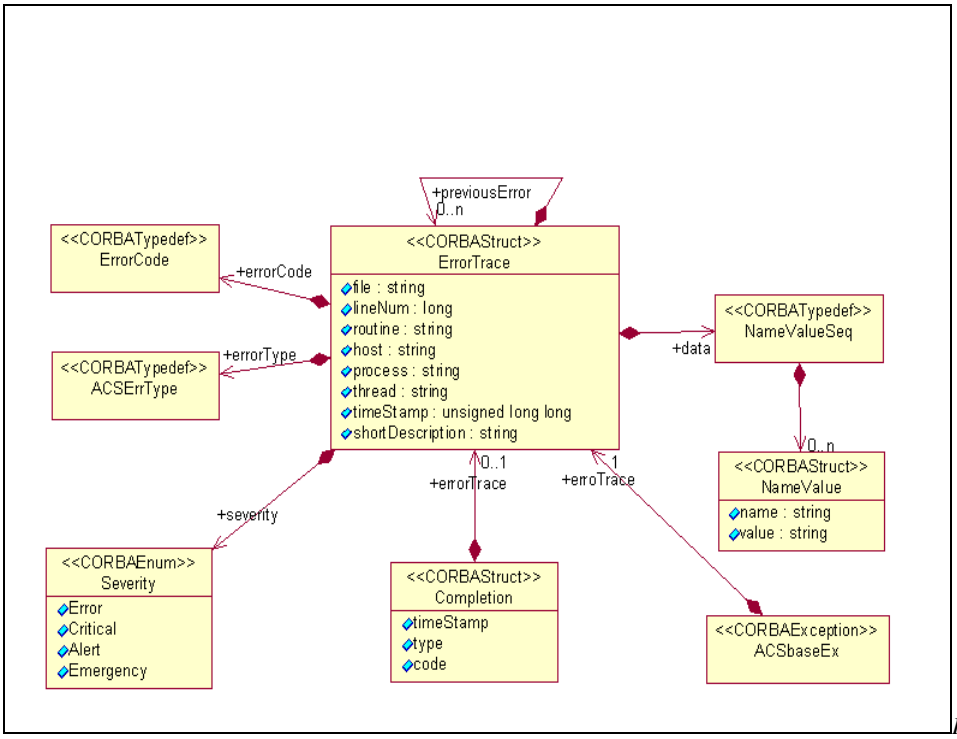


Figure 5 IDL common code UML diagram

All common concepts in ACS error handling are defined in the `acserr.idl` file in the IDL module (namespace) `ACSErr`. Information about a completion/error is described within a structure defined in IDL. Using IDL, we gain two major advantages: error information can be propagated among processes on the same or different machines (interoperability) and error information can be exchanged among processes programmed in different languages (language independence). In addition, entities mapped to a specific language can be used for local error handling.

Completion (which can represent either success or failure) is contained in a structure defined in IDL named `Completion`. When error information is applicable, it has an error backtrace. Error information is also kept in the generated IDL exceptions (see section 5.2), which thus contain error backtraces as well. An error backtrace is defined as a recursive IDL structure, `ErrorTrace`; this means that the structure itself contains a "pointer" to another structure of the same type. In this way, we can see a linked list (within the error backtrace) of all errors propagated by a single exception or completion. Besides a pointer, the structure also contains information about the error itself (error type and error code), all information necessary to clearly identify the context in which the error occurred, severity, and a list of name-value pairs (for details see section 2.2). Here is the definition of the IDL error backtrace structure:

```
struct ErrorTrace
{
   string file;
   long lineNum;
   string routine;

   string host;
   string process;
   string thread;

   unsigned long long timeStamp;

   ACSErrType errorType;
   ACSErr::ErrorCode errorCode;

   ACSErr::Severity severity;

   NameValueSeq data;

   sequence<ErrorTrace, 1> previousError;
};
```

5.2 IDL generated code

Exactly one IDL file is generated per ***type*** (i.e., per XML file). At the level of **type**:

- an IDL exception is generated (within the IDL file) if at least one error code is defined (within the XML file). The naming convention for the IDL exception for a type is: type name suffixed with “Ex”. This exception is used to facilitate an exception hierarchy (i.e., derivation) in programming languages where inheritance is supported (for ALMA this means all three supported languages: C++, Java, and Python).

For each ***code*** (completion and error) there exists:

- an IDL constant with the same name as the completion/error of type ***ACSErr::ErrorCode*** (CORBA ***unsigned long***)

and for the ***error code*** only:

- an IDL exception whose name consists of the name of the error code suffixed with “Ex”, unless the `_suppressExceptionGeneration` attribute is set to ***true*** (the default value is ***false***). This is the case for the last two error codes shown in Section 2.6.

Constants and exceptions are defined within an IDL module (i.e., namespace) that has an identical name to the type, thus avoiding name clashes between codes with the same name but from different types. The name of the module and the name of the IDL file are identical to the name of the type.

There is no special IDL code generated for handling completion/error members. Members are represented in IDL as name-value pairs where the name is the name of the member.

Here is the IDL (`ExmplErrType.idl`) generated from our sample XML:

```
1: #include <acserr.idl>
2:
3: #pragma prefix "alma"
4:
5: module ACSErr {
6: const ACSErr::ACSErrType ACSErrTypeTest = 1200;
7: };
8:
9: module ExmplErrType {
10: const ACSErr::ErrorCode PointingInProgress = 0;
11: const ACSErr::ErrorCode PointingFailure = 1;
12: const ACSErr::ErrorCode PositionNotObtained = 2;
13: const ACSErr::ErrorCode RecordNotFound = 3;
14: const ACSErr::ErrorCode ErrorWOException = 4;
15:
16: exception PointingFailureEx {
17: ACSErr::ErrorTrace errorTrace;
18: };
19: exception PositionNotObtainedEx {
20: ACSErr::ErrorTrace errorTrace;
21: };
22: exception RecordNotFoundEx {
23: ACSErr::ErrorTrace errorTrace;
24: };
25: };//module ExmplErrType
```

The generated IDL file has to be included in all IDL files where completions or errors of the newly generated types will be used (i.e. for asynchronous methods and methods capable of throwing exceptions).

The IDL language does not support a hierarchy of exceptions (**Exception Hierarchy pattern in 1.3**), because inheritance for exceptions is not supported. However, we can implement simple exception hierarchies in the helper classes in the different implementation languages. The mapping is hidden to the programmers by the code generation.

6 C++ implementation

The C++ implementation of the ACS Error System offers two basic features:

- error handling in C++ programs
- error handling over the network using CORBA from C++ programs (i.e sending and/or retrieving remote errors and completions within C++ programs).

It provides functionality for manipulating error backtraces and the structures which contain them, e.g. ***completions*** and ***exceptions,*** and conversion between remote (CORBA) and local structures.

Error information is sent over the network using IDL structures and IDL exceptions, i.e. *just data structures*. Meanwhile, objects (completions and exceptions) are used within C++ (therefore providing the full *"data with functionality"*of the object oriented paradigm).

The C++ implementation of the ACS Error System provides conversion operators between these constructs (i.e. between C++ mapped IDL structures and exceptions and C++ error handling helper classes).

Remember, an error backtrace cannot be sent on its own, but only as a part of a completion structure or an exception.

The C++ implementation provides the following functionality:

- create new completions or exceptions, where an error backtrace is created and filled with run-time and compile-time information
- create new completions or exceptions, adding a previous error backtrace contained within another completion or exception
- send/receive completions or exceptions to/from a remote process (CORBA)
- convert from local helper classes to "remote" CORBA structures and vice versa
- log error backtraces by converting error stack information to strings and XML
- access arbitrary user-defined data(add/get)
- completion/error member support: getter and setter methods
- handlers for uncaught and unspecified exceptions.

The implementation of the error system for C++ consists of:

- ***acserr*** (shared library)

- **generated code** (helper classes)

6.1 acserr library

The `acserr` library contains code that is generated from the ACS error system idl file (acserr.idl), which includes the definitions of ErrorTrace and Completion. It also contains common functionality for use with the generated code, such as: logging, error backtrace manipulation, arbitrary user-defined data manipulation, and the like. The library defines base classes for the generated classes: `CompletionImpl`, `ErrorTraceHelper`, and `ACSbaseExImpl`.

The `ErrorTraceHelper` class contains a reference to the (IDL generated) `ErrorTrace` structure. Thus, there is no need to manipulate the IDL structure in C++ directly.

The `ErrorTraceHelper`, in collaboration with the ACS Logging system, implements a variant of the **Error Handler pattern** in 1.3.

The `ErrorTraceHelper` class provides methods for setting/getting fields of the error backtrace such as `getFileName` and `setError`, to name a few. A description of an error can be obtained using the `getDescription` method. You can add extra information about an error (in the form of name-value pairs) using the `addData` method. The ErrorTrace IDL struct can be extracted (conversion from ErrorTraceHelper to ErrorTrace) with the `getErrorTrace` method. An error backtrace can be logged at any time using the `log` method.

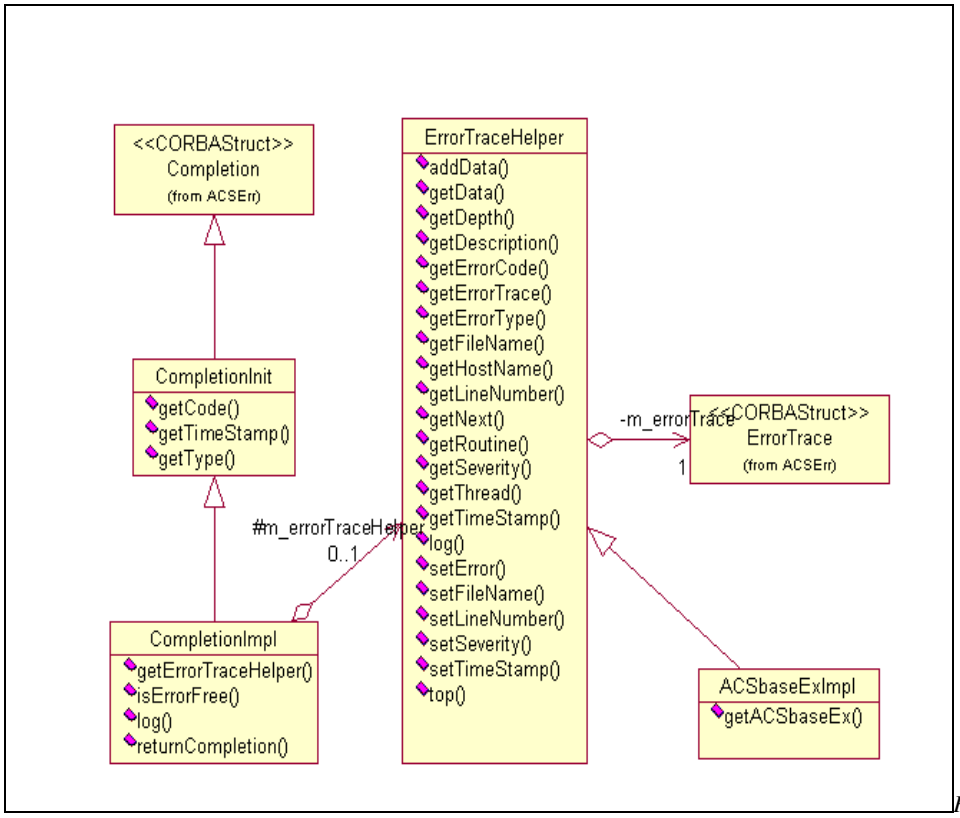


Figure 6 ErrorTraceHelper and CompletionImpl class UML diagrams

`CompletionImpl` derives from `CompletionInit`, thereby including all of `CompletionInit's` functionality and adds:

- `returnCompletion` – conversion from `CompletionImpl` to an IDL `Completion` structure. Used when a completion has to be sent remotely (via CORBA) as a return value.

In general, `ErrorTraceHelper` and `CompletionImpl` classes are not used (i.e. instantiated) directly by developers, but rather through generated code.

The `ACSbaseExImpl` is the (common) base class for all generated exceptions. It cannot be instantiated and thus an exception of type `ACSbaseExImpl` cannot be *thrown*, but an exception of this type can be *caught*.

6.1.1 Handlers for uncaught and unspecified exceptions

The ACS error system for C++ provides handlers for uncaught and unspecified exceptions. Both handlers are installed by the error system initialization, which is done automatically by the container as well as the *`Simple Client`*.

An **unspecified exception** 1.3 means that an exception which was not specified in the method signature was thrown. If a method does not specify that it throws any particular exception(s), it can throw any exception. It is important to know that in C++, unspecified exceptions are detected at run-time and not at compile-time. There are three cases of unspecified exceptions in ACS:

- if the handler catches an unspecified ACS based exception, it reports: *`Unspecified ACS based exception was thrown`* and logs the ACS exception, so that the user can find out from the log where (e.g., file name, line number, etc.) the unspecified exception was thrown.

- if an unspecified CORBA exception is caught the report is: *`Unspecified CORBA exception was thrown.`* In most cases, the reason for this is that a CORBA exception was thrown or propagated in a local (C++) case, where a regular C++ exception should have been thrown.

- If the type of unspecified exception cannot be detected (the exception is neither ACS based nor CORBA based) the message is: *`Unspecified unknown exception was thrown`*

If there is no corresponding *catch statement* for an exception (even though the method signature may have specified that the exception can be thrown), the exception becomes an **uncaught exception** and is caught by the handler. The handler for uncaught exceptions distinguishes between three cases (three types of uncaught exceptions):

- uncaught ACS based exceptions, which are reported as: *`Uncaught ACS based exception`* and the exception is logged

- uncaught CORBA exceptions, which are reported as: *`Uncaught CORBA exception.`* This is usually because there is a missing catch statement for a remote method call.

- uncaught unknown exception, which is reported as: *`Uncaught Unknown exception`*

In both cases (i.e., when an uncaught or an unspecified exception is caught by the handlers), the process is aborted; i.e. the container or simple client is aborted. That means that there is a problem with the code which should be fixed, and there is no sensible way to recover.

6.2 C++ generated code

The XSLT for C++ takes as input an XML error type file with error definitions and outputs generated code for C++. The final result consists of (for each error type):

- a header file

- a shared library containing C++ generated code and code generated from the IDL, e.g. stubs and skeletons (which is why the IDL has to be generated before the C++ code).

The name of the library and the header file is identical to the name of the error type.

The generated header file has to be included where errors of a certain type are going to be caught/thrown, and the generated library must be linked in.

For each type, the following class is generated:

- **type exception class** which is the base class for all exceptions that are generated (for the error codes) for this type. The name of the generated exception class for the type follows the convention: type name suffixed with “ExImpl”. For example, the type ACSErrType implies that a class named `ACSErrTypeExImpl` will be generated.

For each (error) code, the following classes are generated:

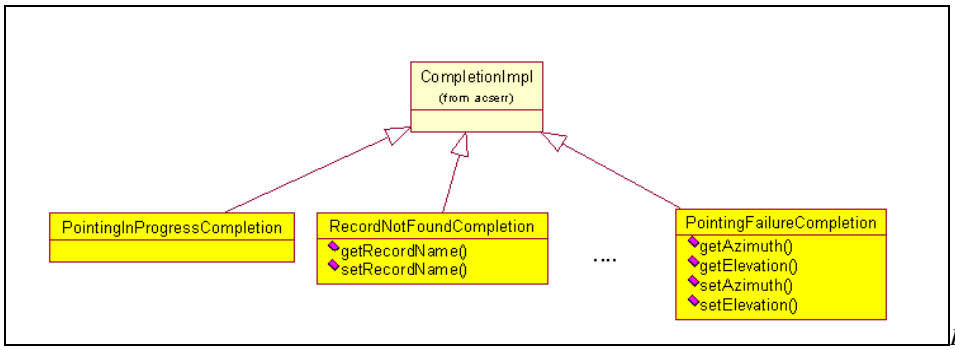


Figure 7

An example diagram for generated completion classes

- one **completion class**. The naming convention for completion classes is the following: the name of the (error) code suffixed with “Completion”. For example, the error code `ACSErrTest0` Completion class would end up being `ACSErrTest0Completion`. All generated completion classes inherit from the common base class `CompletionImpl`, which provides common functionality (see Error: Reference source not found). Generated classes have different constructors to handle different cases, for example the code generated for constructing an erroneous completion is different than code for a successful completion.

- one **exception class.** The naming convention for the exception classes is the name of the error code suffixed with “ExImpl”. For example, the error code `ACSErrTest0` implies that a class named `ACSErrTest0ExImpl` will be automatically generated. Every exception class inherits from the class which is generated for the type (e.g. `ACSErrTestTypeExImpl`) which further derives from the common exception class `ACSbaseImpl` which at the end derives from the `ErrorTraceHelper` class found in the `acserr` library along with the IDL generated exception (i.e., `ACSErrTest0Ex` as an example). The exception that is generated for the *type* cannot be instantiated and thus an exception of this type cannot be thrown (only the exceptions for *error codes* can be instantiated and thrown!). The purpose of generating the (base class) exceptions for types is to simplify catching exceptions, i.e. we can just catch the exception for a particular type and not need to have a catch statement for all of the codes (i.e., exceptions) that belong to that type. For a similar reason, there is also the `ACSbaseExImpl` exception.

For both completion classes and exception classes, if they are generated, there also are generated getter (*get <name of member>*) and setter (*set <name of member>*) methodss for its members.

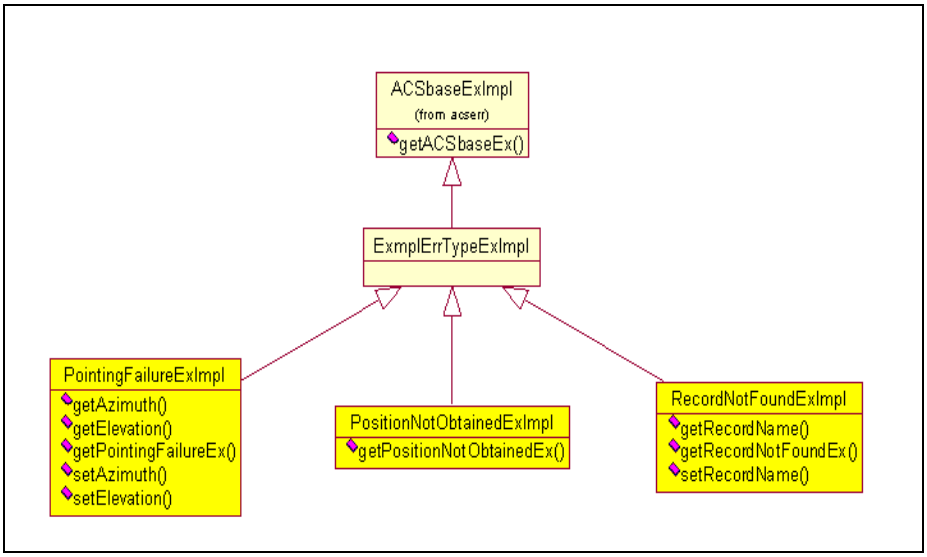


Figure 8 An example diagram for exception generated classes

Classes are generated in a namespace equal to the name of the error type.

6.3 Using Generated C++ Classes

When we want to use the error system in C++ applications, we have to do the following common things:

- Include the generated header file(s)
- Use the appropriate namespace(s)
- link with the generated libraries

6.4 Using Completions in C++

In IDL, one unique general structure for completions is defined, but in C++ a completion is generated for each (error) code.

6.4.1 Using Completions Locally (C++)

The purpose of completion classes in C++ is to use them where native C++ exception handling is inappropriate, e.g. handling errors in asynchronous scenarios, manipulating IDL completion structures, etc. With this approach, we can at any point convert from remote completions to C++ ones and vice versa.

A completion in C++ is just a normal object. If we want to inform someone how an action (method) has completed, we create the proper completion object and transfer it to the caller as a `CompletionImpl`. Completion objects should be transferred as pointers (`CompletionImpl*`) or references (`CompletionImpl&`). Proper memory management for the completion object is necessary; a `CompletionImpl` is just another C++ object.

Below is an example of how a completion can be returned as a pointer. The caller of the method `getRecord(…)` is responsible for the deletion of the `CompletionImpl` object.

```
    //namespace where error classes are defined
    using namespace ExmplErrType;
    CompletionImpl* Example:: getRecord (char *recordName)
    {
      CompletionImpl *comp;
      if (error)
      {
          comp = new RecordNotFoundCompletion(__FILE__, __LINE__,
“Example:: getRecord”);
          comp->setRecordName(recordName);// set value of
RecordName member
      }
      else
          comp = new ACSErrOKCompletion();
      return comp;
    }
```

The caller has to dispose of the *`CompletionImpl`* when it is no longer needed.

The same can be done to return a Completion via a parameter (reference to *`CompletionImpl`*). In this case, the *caller* has to create the *`CompletionImpl`* object and pass it to *`getRecord`* as a parameter.

```
    void Example::getRecord (…, CompletionImpl &comp)
    {
          …
      if (error)
          comp = RecordNotFoundCompletion(__FILE__, __LINE__,
“Example:: getRecord”);
          comp.setRecordName(recordName);
      else
```

```
        comp = ACSErrOKCompletion();
    return;
}
```

In the above examples we can see how to set the value of the error member *RecordName*. An alternate way to do this is to use the generic `addData` method:

```
comp->addData("RecordName", recordName);
```

There are 3 scenarios for instantiating a Completion:

1. creating a successful completion (without an error backtrace)
2. creating an erroneous completion (with error backtrace)
3. creating an erroneous completion (adding a previous error backtrace)

1. Creating successful completions is straightforward - just call the constructor w/o arguments. Here is an example of creating a completion for *ACSErrOK*:

```
…. = new ACSErrTypeOK::ACSErrOKCompletion ();
```

2. When we want to create/start a new error backtrace to be held within a completion, we just create a Completion object. In the constructor we have to provide: file name, line number, and routine name. It is suggested that for file name the *__FILE__* macro is used and for line number, the C++ predefined macro is used: *__LINE__*. These kinds of completion objects are normally created at the point where an error occurred. An error backtrace beginning with the error *RecordNotFoundCompletion* can be instantiated as shown:

```
…. = new ExmplErrType::RecordNotFoundCompletion(__FILE__,
                                    __LINE__,  "Example::
                                    getRecord");
```

An error condition might actually happen in a call to a foreign library, which uses incompatible error handling concepts. In this case it is necessary to wrap the "foreign" error handling into "standard" error handling classes, by following the **Exception Wrapper pattern** in 1.3.

3. The idea behind error backtraces is that error information can be added at any level in the cascade/stack of calls (see section 2.3.1). This can be done by creating a new completion, passing the previous error backtrace as the first parameter. Previous error backtraces can be contained inside any of following types:
   - CORBA (remote) Completion: `Completion*`
   - C++ Completion: `CompletionImpl*`, `<error name>Completion*`

- CORBA exception: `<error name>Ex`
- C++ exception: `<error name>ExImpl`
- raw error backtrace

NOTE: previous errors can only be added to *erroneous* completions.

Here is an example of the creation of a new completion (*PositionNotObtainedCompletion*) adding a previous one (`CompletionImpl*`). First, we have to check if the completion contains an error or not (if it is error free or not). If there is an error, we create a new erroneous completion (*PositionNotObtainedCompletion*) adding the previous one, and return it. An error backtrace from `CompletionImpl` is added to the newly created one in *PositionNotObtainedCompletion*:

```
CompletionImpl* getPosition(…)

{

     CompletionImpl *comp = getRecord(…);

     if (! comp->isErrorFree())

     {

     ACE_CString recordName = comp->getRecordName();

     return new ExmplErrType:PositionNotObtainedCompletion (comp,
                                    __FILE__, __LINE__,
                                    "Example::openFile");

     }

        …

}
```

In the example, we also see how to retrieve the value of the member `RecordName`. An alternative would be to use the generic `getData` method:

```
comp->getData("RecordName");
```

6.4.2 Using Completions Remotely (via CORBA) from C++

Although in *C++* we have a different completion for each specific (error) code, in *IDL* there is just one completion structure which contains a single error backtrace. This IDL structure is used only in remote cases – to send (local) completion information to another (remote) process. So, only data is sent over the network. It is not necessary for the developer to directly manipulate the IDL structure. This is taken care of by the C++

completion classes described in the previous section. The user only has to send and receive the completion.

To send completions over the network, there are two steps:

- define an IDL interface where completions are used
- implement the IDL, where a local completion has to be converted to a remote completion and vice versa.

We can return an IDL completion in a couple of different ways: as a return value or as an (in) out parameter of an IDL operation (for synchronous calls), or using callbacks - the most common way of sending completion (error) information in asynchronous calls. Although completion (error) status is reported using the completion the user has to catch CORBA system exceptions which are used to report CORBA specific errors like network problem, remote object despairing!

6.4.2.1 Synchronous calls

6.4.2.1.1 Completions as a return value

If the pointing method was implemented with a synchronous (blocking) method and a completion (as opposed to an exception) was used for propagating error information, the IDL might look like:

```
interface TelescopeServerSync
{
ACSErr::Completion pointing ();
     …
};
```

That is compiled (mapped) into something that looks like:

```
class POA_TelescopeServerSync: …
{
…
ACSErr::Completion* pointing();
};
```

According to the OMG specification for the IDL to C++ mapping, a variable-length structure (which is what a completion is) is mapped into a C++ class and has to be

returned as a pointer. An implementation of `pointing` (e.g. server side) might then look something like:

```
ACSErr::Completion*

TelescopeServerSyncImpl::pointing()

{

CompletionImpl *er;

// do pointing

if (!error)

{

  er = new ACSErrTypeOK::ACSErrOKCompletion();

}

else

{

  er = new PointingFailureCompletion
                (__FILE__, __LINE__,
                "TelescopeServerSyncImpl::pointing");

  er->setAzimuth(az);

  er->setElevation(el);

 }

return er->returnCompletion();

}
```

Calling `returnCompletion`on a *local* completion is all that has to be done to send local completions of any type remotely, e.g. to convert them from local to remote. It is not necessary to take care of memory management – it is done automatically.

On the client side (where the remote method is invoked), we get a completion object (or more precisely a pointer to one, i.e. a `Completion*`). We *can* deal directly with the retrieved completion (generated from IDL) but it's recommended to convert it into a `CompletionImpl` (which provides error backtrace manipulation functionality and takes care of memory management for us). Be aware that `CompletionImpl` takes over memory management (i.e. deletion) of `Completion`! Here is an example depicting how we can directly convert a remote completion (the `pointing` returns a pointer to the `Completion`) into a `CompletionImpl`, retrieve the value of members: azimuth

and elevation, and then send the completion to the logging system, simply by invoking the *log* method on the completion:

```
CompletionImpl comp =

    telescopeServerSync->pointing();

az = comp.getAzimuth();

el = comp.getElevation();

comp.log();
```

6.4.2.1.2 *Completions as (in)out parameters*

Completions can also be returned as CORBA out parameters. Here is an example.

This is how the IDL should look like:

```
interface ExInterface{

   void someMethod(out ACSErr::Completion comp);

};
```

The implementation:

```
void someMethod(ACSErr::Completion_out comp)

   throw(CORBA::SystemException)

{     //just an example, you can use any kind of completion,
   //Error or non-Error ones

      ACSErrTypeOK::ACSErrOKCompletion c;

   // we have to convert local (C++) completion to CORBA completion

   comp = c.outCompletion(); // if c was allocated on the heap
   //… and we wanted that c is deleted we would have to pass
   //… true as parameter to the method.

}
```

Finally, the client:

```
ACSErr::Completion_var comp_var;

component->someMethod(comp_var.out());

CompletionImpl comp = comp_var;//just to use the convenience methods

comp.log();
```

6.4.2.2 Asynchronous calls

An asynchronous method accepts a Callback as a parameter. The caller will pass within this parameter the instance of the Callback object to be used to asynchronously communicate the outcome of the request. In most cases one can use one of the already existing Callback types predefined in baci (1.3).

Here is an example showing how the asynchronous `pointing` method of the TelescopeServer, from the example, might look like in its IDL specification:

```
interface TelescopeServer

{

void pointing(in ACS::CBvoid cb);

      …

};
```

The `pointing` method has to be called by passing a callback object. In our case a predefined callback from baci, *`CBvoid`*, is used, but a new one could be defined. Here is how *`CBvoid`* from baci is defined:

```
interface CBvoid : Callback {

oneway void working (in Completion c, in CBDescOut
                              desc);

oneway void done (in Completion c, in CBDescOut desc);

};
```

The implementation of the pointing method has to be done in such a way that it stores the callback object (i.e., a reference to it), starts the pointing procedure (e.g., in a separate thread), and then returns.

The pointing procedure (in its own thread) has to use the remote callback object to notify the caller when and how it has finished. It passes a completion as a parameter to the `working` or `done` methods. During the execution of the pointing procedure, the client has to be notified that the operation is still in progress; this is done by invoking the `working` method with the completion (`PointingInProgressCompletion`).

Here is an example of how the notification could be done: first we create (shown) and fill (not shown) the *`CBDescIn`* structure. Then we create a *`PointingInProgress`* completion. Finally, we invoke the *`working`* method on the (remote) callback object, passing the completion as the first parameter. Because *`PointingInProgressCompletion`* derives from *`Completion`*, we can directly pass it as a parameter to the working method. Here is a snippet of code which shows what the server's pointing method might look like (portions omitted):

```
CBDescIn cbDescIn;

                …

PointingInProgressCompletion comp(__FILE__, __LINE__, "…");

cb->working(comp, cbDescIn);
```

6.5 Using exceptions in C++

Exceptions in C++ are a very useful mechanism for handling errors, but there are a few potential pitfalls that the user has to be aware of. How to use exceptions in C++, what the user has to be aware of, and possible problems/pitfalls when using exceptions are covered in detail in references 1.31.31.31.3, which are recommended reading for all C++ developers dealing with exceptions. Here are, in short form, a few recommendations for dealing with exceptions in C++:

- release all resources that cannot be automatically released *before* throwing an exception (i.e., free up memory allocated on the heap and such)
- throw an exception as an object, not as a pointer to the object
- catch an exception as an object reference, and not as a pointer or as copy of an object
- be aware that unspecified exceptions in C++ are detected at run-time and not at compile-time
- try to avoid using generic catch staments: `catch(…)`

6.5.1 Using local exceptions (C++)

As mentioned, for each type a C++ (exception) class is generated (type exception). The type exception inherits from C++ (exception) classes that are generated for error codes of that type. The developer just needs to instantiate the exception object corresponding to a certain error code. It is important to note that the (base) type exception cannot be instantiated and exists just to ease the work of catching exceptions.

Instantiation of exception objects is very similar to the instantiation of a completion object. There are two cases:

1. creating a new exception and beginning the error stack
2. creating a new exception and adding a previous exception or completion of any type, adding to the error backtrace

1. Here is an example of the first case:

```
throw ExmplErrType::RecordNotFoundExImpl(__FILE__,  __LINE__,
                          "DatabaseServerImpl::getRecord");
```

This is similar to the record-can-not-be-found example shown previously, but an exception is used instead of a completion. The exception is created by providing the following parameters to the constructor: file name (*__FILE__* predefined macro should be used), line number (*__LINE__* predefined macro should be used) and routine name ("DatabaseServerImpl::getRecord" in this example).

2. If we want to add a previous exception or completion with an error backtrace, we can do so as follows:

```
catch (RecordNotFoundExImpl &ex)
{
throw PositionNotObtainedExImpl(ex, __FILE__, __LINE__,
                          "starServerImpl::getPosition");
}
```

Catching ACS exceptions has to be done in the standard C++ way. Since we have a hierarchy of exceptions (see Figure 8) we can catch exceptions at different levels:

- at the level of ACS (we can catch any ACS exceptions):
  `catch (ACSbaseExImpl &ex)`
- at the level of type (we can catch any exception of a certain type):
  `catch (ExmplErrTypeExImpl &ex)`
- at the level of error code, for example:
  `catch (RecordNotFoundExImpl &ex)`

6.5.2 Using remote (CORBA) exceptions from C++

An important difference between local and remote (CORBA) exceptions is that CORBA user exceptions and consequently CORBA ACS exceptions do not support inheritance. Thus, there cannot be a hierarchy for generated CORBA exceptions (see Figure 8). If we want to "simulate" an exception hierarchy for remote cases, we can do this by specifying exceptions for all (or some) levels: the common base exception *ACSBaseEx*, the type level exception, or/and the code level exception.

If we want to throw exceptions from IDL methods, we have to specify this in the IDL using the `raises` keyword, specifying the generated exceptions that can be thrown (as in the following example for the `getRecord` method of DatabaseServer):

```
…
void getRecord (…)
      raises (ExmplErrType::ExmplErrTypeEx,
              ExmplErrType::RecordNotFoundEx);
```

```
        …
```

Here the implementation of the *getRecord* method can throw an exception at the level of type: *ExmplErrTypeEx* or an exception at the level of error code: *RecordNotFoundEx*. If the method threw a common base exception then the exception *ACSBaseEx* would have to be added as well.

In the implementation of the interface, we throw exceptions by:

- first create (or catch) an exception, similar to the local case and then
- call the get<NameOfRemoteException>() method on that object to get the remote exception, which can be thrown.

Be aware that the name of local and remote exceptions differs only in the suffix *Impl*. The implementation for the above example might look like:

```
void DatabaseImpl::getRecord (…)
      throw ( CORBA::SystemException,
              ExmplErrType::ExmplErrTypeEx,
              ExmplErrType::RecordNotFoundEx)

{
              …

throw ExmplErrType::RecordNotFoundExImpl (
      __FILE__, __LINE__,
      "DatabaseServerImpl::getRecord"
      ).getRecordNotFoundEx();

              …

}
```

If we want to instantiate a new exception adding a previous one, we do this as in the local case described above, and then eventually "convert" to a remote exception.

On the client side where the exception has to be caught, we have to be careful with remote exceptions – the client must catch exception RecordNotFoundEx and not RecordNotFoundExImpl. Exceptions (remote) caught in such a way can be wrapped into a corresponding local (RecordNotFoundExImpl) one or can be added to a newly created one as is done in the local case.

6.5.3 Combining completions with exceptions

It is also possible to combine completions with exceptions. In general there are two cases:

- we have a completion, but we would like to throw an ACS exception
- we have an ACS exception (local or remote) and we would like to return a completion.

The “conversion,” which is trivial (and is in principal the same for both cases), works in such a way that a new exception/completion is created using the previous completion/exception passed as the first parameter, in the same way as it is done for creating a backtrace. In fact, the backtrace is created. There is no direct conversion from an ACS exception to completion or vice verse (an intentional design decision), which forces the user to build an errortrace at each level of call (i.e., to add context at each level).

Here is an example, where the `getPostion` method throws a CORBA exception (`RecordNotFoundEx`) if it encounters an error. After it is caught, the exception is passed to the constructor of the completion (`PositionNotObtainedCompletion`):

```
try

{

    getPosition(…)

}

catch (RecordNotFoundEx &ex)

{

PositionNotObtainedCompletion *comp =
     new PositionNotObtainedCompletion(ex, __FILE__,
                  __LINE__, "starServerImpl::getPosition");

}
```

The next example covers the case where we get a completion (in this case from a remote call, but the code would be similar for a local call) and we want to throw an exception:

```
CompletionImpl comp = getRecord(…);

If (!comp.isErrorFree())

{
```

```
    throw PositionNotObtainedExImpl(comp, __FILE__,
                                    __LINE__,

                            "starServerImpl::getPosition");

}
```

7 Python Implementation

The Python implementation of the ACS Error System provides the same core functionality of the C++ error system. At the most basic level, the C++ and Python implementations are nearly identical: the developer defines an XML file that is a valid instance of the *ACSError* schema and then edits the *Makefile* accordingly. From there, a Python module (named identically to the error type with "Impl" appended to it) using ACS Error System helper classes is automatically generated by the "all" target of the *Makefile*. The helper classes have get and set methods. **For specific information on the methods available to the developer from the exception and completion helper classes, see the pydoc for *ACSErr* in the *Acspy.Common.Err* package.**

7.1 Raising and Catching Exceptions

The following example is the Python equivalent of sections 6.4 and 6.5:

```
1: import ExmplErrType
2: import ExmplErrTypeImpl
3:
4: try:
5:    raise ExmplErrTypeImpl.RecordNotFoundExImpl()
6: except ExmplErrType.RecordNotFoundEx, e:
7:     helperObject = ExmplErrTypeImpl.RecordNotFoundExImpl(exception=e,
    create=0)
8:    helperObject.log()
```

1 Import the IDL stubs for the IDL file generated by the ACS Error System XML Makefile target. This is only being done so we can show that the exception helper class (line 5) is actually implementing the IDL exception type (line 6).

2 Import the Python module generated by the ACS Error System framework. This includes implementations for all exceptions defined in the generated IDL file (see line 1) as well as completion helper classes.

4-8 In this simple yet complete example we: raise a local generated exception, catch it as if it were a remote IDL exception (i.e., no helper methods are available for object *e* on line 6), and then convert it back into a local exception without adding any new error information.

5 An exception generated from the ACS Error System framework is thrown. In Python, there is no difference between local and remote cases and the generated exception class takes care of everything for the developer.

6 Here the exception thrown on line 5 is caught as if it were really a remote CORBA exception that was being thrown. Obviously we could have just caught the *…ExImpl* exception directly but this would only work in local cases. Please note that object *e* would not have access to any of the nice exception helper methods such as *log* in the remote case.

7 The IDL exception instance e is reconverted into an ACS Error System helper exception by specifying the *exception* keyword parameter. The *create* keyword parameter is set to *0* to ensure no new error information is appended to *helperObject*'s error stack (like line 7 for example). *e* is reconverted into an ACS Error System helper exception so a number of methods defined only in Python become available to developers (see line 8).

8 The *log* method is obviously not defined for the IDL *RecordNotFoundEx* exception. It's available only because of line 7.

7.2 Creating and Using Completions

This trivial example shows how Python Completion helper classes generated by the ACS Error System can be used. **Of particular interest to developers working with pure-CORBA *ACSErr::Completion* objects is the *addComplHelperMethods*** function found in the *Acspy.Common.Err* package. This function dynamically attaches helper methods (see the pydoc for *Acspy.Common.Err* for all descriptions and signatures) similar to those of ACS-generated exceptions to pure-CORBA Completion objects (e.g., the *Completion out* parameter of the *get_sync* method of BACI properties).

```
1: import ExmplErrTypeImpl
2:
3: def someFunc():
4:   #...
5:   return ExmplErrTypeImpl.RecordNotFoundCompletionImpl()
6:
7: helperObject = someFunc()
8: helperObject.log()
```

1 Import the Python module generated by the ACS Error System framework. This includes completion helper classes for all types of errors.

3-5 *someFunc* is just a function which always returns a Completion IDL struct.

5 By using the *RecordNotFoundCompletionImpl* class derived from the Completion CORBA stub, we do not have to worry about filling out individual fields of the Completion IDL struct. The infrastructure takes care of this automatically.

7 Obtain a local Completion. If this were remote (i.e., a strict IDL Completion struct instance), it could be converted into the *RecordNotFoundCompletionImpl* class using the *create* keyword argument supplied to *RecordNotFoundCompletionImpl*'s constructor (as seen in the previous section).

8 The *log* method is obviously not defined for the IDL Completion struct. It's available because *helperObject* is actually an instance of a Completion helper class.

7.2.1 Completions as out parameters

Completions can also be passed to the server as *out* parameters. In Python the implementation is somewhat unusual. The *out* parameters are just returned. If there are more than one, return them as a tuple. If there is also a return value, it is the first member of the tuple. *Inout* parameters should be passed as arguments, as if they were *in* parameters, and received as a return value as if they were *out* parameters. Here is an example.

This is how the IDL should look like:

```
interface ExInterface{

   void someMethod(out ACSErr::Completion comp);

};
```

The implementation,:

```
def someMethod(self):

return ACSErrTypeOKImpl.ACSErrOKCompletionImpl()
```

Finally, the client:

```
comp = component.someMethod()

addComplHelperMethods(comp)

comp.log()
```

8 Java Implementation

8.1 Introduction.

As Java developers, we are used to having an error system that has an inheritance structure and also a linking structure from which we can see what caused our error. Unfortunately CORBA does not provide this functionality, so ACS has developed a method to provide it. The Java implementation of the ACS Error System takes advantage of the exception hierarchy provided by the Java language's error system API. Java

provides chained-exception functionality which allows developers to create exceptions using other exceptions. For example:

```
npe = new NullPointerException();
ex = new Exception(npe);
```

This type of exception handling results in a hierarchy of exceptions. This hierarchical structure is provided for the exception types and for the linking of exceptions which *caused* the exception. For clarity, an exception of type NullPointerException in Java has the inheritance hierarchy

NullPointerException
(is a)
(
RuntimeException
(is a)
(
Exception
(is a)

Throwable

The same NullPointerException also has a *caused-by* linking structure:

NullPointerException
(was caused by)
(
ClassNotFoundException
(was caused by)
(
ClassCastException

This type of exception *caused-by* linking can be seen during runtime when any exception is caught and its *printStackTrace()* is displayed. Another way to see what has caused an exception is to use a method called *getCause().*

CORBA does not provide a way (natively) to transport these types of linked error trace structures. However, this functionality is provided by the *ErrorTrace* struct defined in *acserr/ws/idl/acserr.idl* (see Section 5.1).

The main goal of implementing the ACS Error System in Java differently from the direct CORBA mappings, as done in C++, is to avoid dealing directly with the *ErrorTrace.* To facilitate the passing of the error hierarchies over CORBA, ACS provides a class, *AcsJException*, which extends the Java *Exception* class. This ACS class encapsulates the functionality developers need to convert an exception (local) into an object that can be transported via CORBA (remote).

8.2 Generated wrapper classes.

When a new type of error is created and compiled, not only is the IDL file generated (see Section 5.2) but a set of wrapper classes are also generated. These wrapper classes extend the AcsJException class, provided by the ACS Error System. The advantage of the wrapper classes is that they hide the *ErrorTrace* details from the user. See Appendix C for the AcsJException public methods.

Developers are not *required* to use the AcsJ- wrapper classes but the classes are generated to help with converting the errors into COBRA-friendly objects. However, after you look through their implementation, you'll be happy they are generated and will probably *want* to use them!

8.3 Let's try an example!

First we need to create a new Error Type. Let's call it ErrorSystemExample. Below is the XML file that defines the error codes (See Section 4 for more information about how to add new error types).

```
<?xml version="1.0" encoding="UTF-8"?>
<Type xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:noNamespaceSchemaLocation="ACSError.xsd"  name="ErrorSystemExample"
        type="13" _prefix="alma">

      <!-- these are the codes for the ERRORS -->
       <ErrorCode name="NothingCanBeScheduledError"
            shortDescription="Error in Scheduling System"
            description="Nothing could be scheduled error"/>

       <ErrorCode name="PipelineProcessingRequestError"
            shortDescription="Error with pipeline processing request"
            description="1"/>
</Type>
```

There are two error codes defined above as part of the ErrorSystemExample type:

- NothingCanBeScheduledError
- PipelineProcessingRequestError

As described in Section 3, if the Makefile tag ACSERRDEF has been defined then an IDL file will be generated from the XML file.

If the XML file is called *ErrorSystemExample.xml* then the Makefile tag should be set as

ACSERRDEF = ErrorSystemExample

Then the *make all* target will produce and compile the IDL file *ErrorSystemExample.idl.* (see Section 5.2) as shown below.

```
 #ifndef  _ErrorSystemExample_IDL_
#define  _ErrorSystemExample_IDL_
#include <acserr.idl>

#pragma prefix "alma"

module ACSErr {
	// type
	const ACSErr::ACSErrType ErrorSystemExample = 13;
};

module ErrorSystemExample {

	 const ACSErr::ErrorCode NothingCanBeScheduledError = 0;
	 const ACSErr::ErrorCode PipelineProcessingRequestError = 1;

	// exception for type:
	exception ErrorSystemExampleEx {
		ACSErr::ErrorTrace errorTrace;
	};

	// exceptions for codes:
	exception NothingCanBeScheduledErrorEx {
		ACSErr::ErrorTrace errorTrace;
	};

	except ion PipelineProcessingRequestErrorEx {
		ACSErr::ErrorTrace errorTrace;
	};

};//module

#endif
```

The wrapper classes that get generated for this XML/IDL are

- *AcsJErrorSystemExampleEx.class*, which is *abstract* and *extends AcsJException*
- *AcsJNothingCanBeScheduledErrorEx.class*, which *extends AcsJErrorSystemExampleEx*

- *AcsJPipelineProcessingRequestErrorEx.class*, which *extends AcsJErrorSystemExampleEx*

Note: As you may recall, you can specify the *DEBUG=on* tag in your Makefile to get the source files for these generated classes (added to the jar file that is produced).

Now once the types and error codes are created, you can use them in a component.

The following is a simple component that will throw our errors defined above over CORBA.

Below is the IDL interface for our component: *ErrorSystemComponent.idl*

```
#ifndef _ERRORSYSTEMCOMPONENT_IDL
#define _ERRORSYSTEMCOMPONENT_IDL

#include <acscomponent.idl>
#include <ErrorSystemExample.idl>

#pragma prefix "alma"

module demo
{
   interface ErrorSystem : ACS::ACSComponent {

   void tryToScheduleSomething()
            raises (ErrorSystemExample::NothingCanBeScheduledErrorEx);
   void tryToProcessSomething()
            raises (ErrorSystemExample::PipelineProcessingRequestErrorEx);
   void usingWrapperClasses1()
            raises (ErrorSystemExample::NothingCanBeScheduledErrorEx);
   void usingWrapperClasses2();

   };
};
#endif
```

Th implementation source file is: *ErrorSytemImpl.java*, which we will explain in segments.

The package this class is in and the imports it requires:

```
package alma.demo.ErrorSystemComp;

import alma.acs.component.ComponentImplBase;
import alma.demo.ErrorSystemOperations;
import alma.ErrorSystemExample.ErrorSystemExampleEx;
```

```
import alma.ErrorSystemExample.NothingCanBeScheduledErrorEx;
import alma.ErrorSystemExample.PipelineProcessingRequestErrorEx;
import alma.ErrorSystemExample.wrappers.AcsJErrorSystemExampleEx;
import alma.ErrorSystemExample.wrappers.AcsJNothingCanBeScheduledErrorEx;
import alma.ErrorSystemExample.wrappers.AcsJPipelineProcessingRequestErrorEx;
```

Class header and constructor:

```
public class ErrorSystemImpl extends ComponentImplBase
                                        implements ErrorSystemOperations {

   public ErrorSystemImpl() {}
```

This method below simulates getting a null scheduling block and trying to schedule it. Because the scheduling block is null, the method throws a NullPointerException which is caught in the same method. When that error is caught, the generated wrapper class for the NothingCanBeScheduledErrorEx is created using the NullPointerException. We do this because we want the method to throw a NothingCanBeScheduledErrorEx that can be transported over CORBA. So in the wrapper class our NothingCanBeScheduledErrorEx is created with the ErrorTrace. We can then throw the remote exception.

```
   public void tryToScheduleSomething() throws NothingCanBeScheduledErrorEx {
      String schedblock = null;
      try
      {
         if(schedblock != null) {
            //Schedule it!
         } else {
            throw new NullPointerException("Can't scheduled a null SB!");
         }
      }
      catch (NullPointerException e)
      {
         //Create the wrapper class for the NothingCanBeScheduledErrorEx using the
         //exception that was caught.
         AcsJNothingCanBeScheduledErrorEx e2 =
                                 new AcsJNothingCanBeScheduledErrorEx(e);
         // use the method in the generated wrapper class to prepare our Java exception
         // for CORBA transport. Basically allowing all the information of what it is
and
         // where it came from and why it happened to be retained after it's been sent
         // over CORBA
         throw e2.toNothingCanBeScheduledErrorEx();
      }
   }
```

This next method simulates how an error might be thrown if the Pipeline could not process something. Logically, if something needs to be processed it must have been scheduled first. This is why we're using the method above. Since nothing was scheduled, we get the NothingCanBeScheduledErrorEx from something trying to be scheduled. Like in the method above, the generated wrapper class is used to create the error we want to throw from this method.

```
  public void tryToProcessSomething() throws PipelineProcessingRequestErrorEx {
     //Trying to process something.
     try {
        // To process something there must have been something scheduled.
        // Since there was nothing scheduled the Nothing can be scheduled
        // error was thrown
        tryToScheduleSomething();

     } catch(NothingCanBeScheduledErrorEx e) {
        //Create one of the wrapper-classes using the caught exception.
        AcsJPipelineProcessingRequestErrorEx e2 =
                        new AcsJPipelineProcessingRequestErrorEx(e);

        // use the method in the generated wrapper class to prepare our Java exception
        // for CORBA transport. Basically allowing all the information of what it is
and
        // where it came from and why it happened to be retained after its been sent
        // over CORBA
        throw e2.toPipelineProcessingRequestErrorEx();
     }
  }
```

Now a common thought for Java developers might be: "Since my error code, *NothingCanBeScheduledError*, in xml is of type, *ErrorSystemExample*, I should be able to catch the *ErrorSystemExample* when *NothingCanBeScheduledError* is thrown". This is unfortunately ***not*** the case. Since CORBA does not provide exception inheritance support for this scenario is not possible. However it is possible to do this if you are using the generated wrapper-classes as seen in the example method below.

```
  public void usingWrapperClasses1() throws NothingCanBeScheduledErrorEx {
     //In this method I want to show how you can use the base-class
     //wrapper-class
     try {
        throw new AcsJNothingCanBeScheduledErrorEx(
                        "Couldn't scheduled due to bad weather!");
     } catch(AcsJErrorSystemExampleEx e){
        throw (
(AcsJNothingCanBeScheduledErrorEx)e).toNothingCanBeScheduledErrorEx();
     }
```

```
    }
```

Likewise, if you just want the *printStackTrace()* from the exception you could do something like the following method.

```
    public void usingWrapperClasses2() {
        //In this method I want to show how you can use the base-class
        //wrapper-class
        try {
            throw new AcsJNothingCanBeScheduledErrorEx("Testing
printStackTrace!");
        } catch(AcsJErrorSystemExampleEx e){
            e.printStackTrace();
        }
    }

} //Closing Class bracket.
```

And just to stress this point to its limit since all AcsJ- wrapper classes extend from the general *Exception* class from Java you could just catch that one.

And just for reference, the generated helper source file, *ErrorSystemHelper.java* is shown below.

```
package alma.demo.ErrorSystemComp;

import java.util.logging.Logger;
import alma.acs.component.ComponentLifecycle;
import alma.acs.container.ComponentHelper;
import alma.demo.ErrorSystemOperations;
import alma.demo.ErrorSystemPOATie;
import alma.demo.ErrorSystemComp.ErrorSystemImpl;

public class ErrorSystemHelper extends ComponentHelper
{
  public ErrorSystemHelper(Logger containerLogger)
  {
       super(containerLogger);
  }

  protected ComponentLifecycle _createComponentImpl()
  {
       return new ErrorSystemImpl();
  }

  /**
  * @see alma.acs.container.ComponentHelper#_getPOATieClass()
```

```
 */
 protected Class _getPOATieClass()
 {
      return ErrorSystemPOATie.class;
 }

 /**
 * @see alma.acs.container.ComponentHelper#getOperationsInterface()
 */
 protected Class _getOperationsInterface()
 {
      return ErrorSystemOperations.class;
 }

}
```

8.4 Another example.

The source files for the next example, described below, can be found in the jcontexmpl module from ACS, in the XmlComponent sub-module.

In this example our error type is ACSErrTypeTest and we have defined 6 error codes. We will only see XmlComponentError used in the below implementation.

```
 <?xml version="1.0" encoding="UTF-8"?>
 <Type xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:noNamespaceSchemaLocation="ACSError.xsd"
        name="ACSErrTypeTest"
        type="12"
        _prefix="alma">

     <Code name="ACSErrTestOK" shortDescription="Test OK" description="Test OK"/>

      <ErrorCode name="XmlComponentError"
                  shortDescription="Error in XmlComponent"
                  description="Error in XmlComponent (from the ACS example module
                   jcontexmpl)"/>

        <ErrorCode name="ACSErrTest1" shortDescription="Test error 1"
                                       description="Test error 1"/>
        <ErrorCode name="ACSErrTest2" shortDescription="Test error 2"
                                      description="Test error 2"/>
        <ErrorCode name="ACSErrTest3" shortDescription="Test error 3"
                                       description="Test error 3"/>
        <ErrorCode name="ACSErrTest4" shortDescription="Test error 4"
                                       description="Test error 4"
                                       suppressExceptionGeneration="1"/>
        <ErrorCode name="ACSErrTest5" shortDescription="Test error 5"
```

```
                                        description="Test error 5" _
                                        suppressExceptionGeneration="true"/>
  </Type>
```

The IDL compiler then generates the IDL file *ACSErrTypeTest.idl* shown below.

```
#ifndef  _ACSErrTypeTest_IDL_
#define  _ACSErrTypeTest_IDL_

#include <acserr.idl>

#pragma prefix "alma"

module ACSErr {
      // type
      const ACSErr::ACSErrType ACSErrTypeTest = 12;
};

module ACSErrTypeTest {
       const ACSErr::ErrorCode ACSErrTestOK = 0;
       const ACSErr::ErrorCode XmlComponentError = 1;
       const ACSErr::ErrorCode ACSErrTest1 = 2;
       const ACSErr::ErrorCode ACSErrTest2 = 3;
       const ACSErr::ErrorCode ACSErrTest3 = 4;
       const ACSErr::ErrorCode ACSErrTest4 = 5;
       const ACSErr::ErrorCode ACSErrTest5 = 6;

      // excption for type:
      exception ACSErrTypeTestEx {
            ACSErr::ErrorTrace errorTrace;
      };
      // excptions for codes:
      exception XmlComponentErrorEx {
            ACSErr::ErrorTrace errorTrace;
      };
      exception ACSErrTest1Ex {
            ACSErr::ErrorTrace errorTrace;
      };
      exception ACSErrTest2Ex {
            ACSErr::ErrorTrace errorTrace;
      };
      exception ACSErrTest3Ex {
            ACSErr::ErrorTrace errorTrace;
      };
};//module
```

This file goes into the *lib/ACSErrTypeTest.jar* file and into the package *alma.ACSErrTypeTest.*

Below is an excerpt from the IDL file *XmlComponent.idl,* which is the IDL file that describes the interface for the XmlComponent. You can see that the exception defined above is used.

```
void exceptionMethod() raises (ACSErrTypeTest::XmlComponentErrorEx);
```

Then the generated wrapper class for this example would be *AcsJXmlComponentErrorEx.* This is the class that extends the wrapper class for its base class, *AcsErrTypeTest* which extends *AcsJException* and provides support for easy transportation over CORBA. The code fragments shown below demonstrate how these exceptions can be used. The source code can be found in *XmlComponentImpl.java*

```
/**
 * At the CORBA interface level, we must use the CORBA-exceptions.
 * throws XmlComponentErrorEx with an alma.ACSErr.ErrorTrace inside;
 * the ErrorTrace will contain a NullPointerException.
 */
public void exceptionMethod() throws XmlComponentErrorEx  {
      try {
            // an internal method that works with native Java exceptions
            internalExceptionMethod();
      } catch (AcsJXmlComponentErrorEx e) {
            // convert to CORBA-compatible exception
            throw e.toXmlComponentErrorEx();
      }
}
```

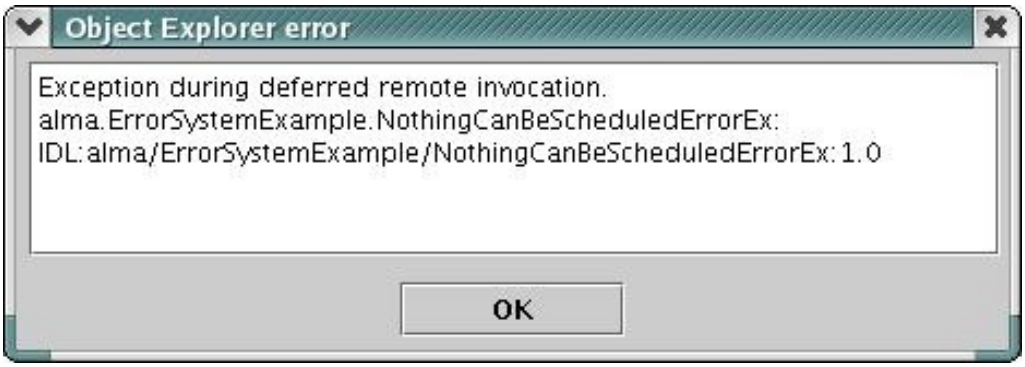
```
/**
 * Inside the Java implementation, we can throw around
 * native Java exceptions generated from the error specifications.
 * throws AcsJXmlComponentErrorEx because that one is easier to
 * work with than the corresponding XmlComponentErrorEx, and
 * this method is not part of the (CORBA) component interface.
 */
public void internalExceptionMethod() throws AcsJXmlComponentErrorEx {
      try {
            // do something that can throw an exception
            throw new NullPointerException("dirty NPE for testing...");
      } catch (NullPointerException npe) {
            // this shows how native Java exceptions can be daisy-chained
            throw new AcsJXmlComponentErrorEx("use me for container tests", npe);
      }
}
```

In the above example an AcsJXmlComponentErrorEx is thrown. It is caused by a NullPointerException which was possibly caused by one or more other exceptions. To send this chained exception over CORBA, it must be converted to

XmlComponentErrorEx. This can be done using the method *toXmlComponentErrorEx()* provided in the wrapper classes.

8.5 How to test the above examples.

The best way to test these examples is to use object explorer. The thing to look for is when the pop up window appears stating that a CORBA error has occurred; look at the error type that is shown, and you should see the error which the example defined. For example, if you invoked *method1()* from the first example you would see the following image showing a *NothingCanBeScheduledError.*



For the *usingWrapperClasses2()*method, from Section 8.3, you will see the *printStackTrace()* in the output from the container.

```
alma.ErrorSystemExample.wrappers.AcsJNothingCanBeScheduledErrorEx: Testing
printStackTrace!
  at
alma.demo.ErrorSystemComp.ErrorSystemImpl.usingWrapperClasses2(ErrorSystemImpl.jav
a:78)
  at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
  at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
  at
sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
  at java.lang.reflect.Method.invoke(Method.java:324)
  at alma.acs.container.ContainerSealant.invoke(ContainerSealant.java:115)
  at $Proxy0.usingWrapperClasses2(Unknown Source)
  at alma.demo.ErrorSystemPOATie.usingWrapperClasses2(ErrorSystemPOATie.java:43)
  at alma.demo.ErrorSystemPOA._invoke(ErrorSystemPOA.java:46)
  at org.jacorb.poa.RequestProcessor.invokeOperation(Unknown Source)
  at org.jacorb.poa.RequestProcessor.process(Unknown Source)
  at org.jacorb.poa.RequestProcessor.run(Unknown Source)
```

8.6 Error types & Exceptions in Java.

An important thing to note: The XML definition of your error type implies that the error codes are of that particular type. After it is compiled to IDL, the exceptions are limited by CORBA's no exception inheritance limitation. So basically, once your error codes are

compiled to IDL they have no super class. However, as mentioned in Section 8.3 the AcsJ- wrapper classes do have this capability.

8.7 Advantages of Java style exception handling.

There are three main advantages of using the ACS Error System when sending exceptions through CORBA. These include:

- It makes use of polymorphism,
- It is quite easy to work with,
- It easily integrates with non-ACS Java exceptions.

8.8 Completions

8.8.1 Completions as out parameters

Completions can also be passed to the server as out parameters. Here is an example.

This is how the IDL should look like:

```
interface ExInterface{

   void someMethod(out ACSErr::Completion comp);

};
```

The implementation:

```
public void someMethod(alma.ACSErr.CompletionHolder comp){

   comp.value=(new ACSErrOKAcsJCompletion()).toCorbaCompletion();

}
```

Finally, the client:

```
alma.ACSErr.CompletionHolder comp = new
alma.ACSErr.Completionholder();

try{

   component.someMethod(comp);

}catch(Throwable th){//manage the exception

}

alma.ACSErrTypeOK.wrappers.ACSErrOKAcsJCompletion c= comp.value;
```

Appendix A. *ACSError* schema

The latest documentation for the ACSError xml schema is available online:

http://www.eso.org/projects/alma/develop/acs/OnlineDocs/ACS_docs/schemas/Alma_ACSError/index.html

Appendix B. Type allocation

Type numbers from 0 to 1000 are used and/or reserved by ACS itself. Here are the reserved type numbers for other alma subsystems:

- **ACS** 0 – 9999
- **CONTROL** 10000 – 19999
- **CORRELATOR** 20000 – 29999
- **OFFLINE** 30000 – 39999
- **TELCAL** 40000 – 49999
- **PIPELINE** 50000 – 59999
- **ARCHIVE** 60000 – 69999
- **EXECUTIVE** 70000 – 79999
- **SCHEDULING** 80000 – 89999
- **OBSPREP** 90000 – 99999
- **HLA** 100000 – 109999
- **ACA** 110000 – 119999
- **Examples/Test** 900000 – 909999

You can also find the type allocation online at:
http://almasw.hq.eso.org/almasw/bin/view/HLA/CompletionErrorTypes

Appendix C. AcsJException Public Methods

Public Member Functions *(See ACS API for full API)*

AcsJException ()
AcsJException (String message)
AcsJException (String message, Throwable cause)
AcsJException (Throwable cause)
AcsJException (ErrorTrace etCause)
AcsJException (String message, ErrorTrace etCause)
AcsJException (ACSException cause)
AcsJException (String message, ACSException cause)
abstract UserException toCorbaException ()
void setSeverity ()
Severity getSeverity ()
Object setProperty (String key, String value)
String getProperty (String key) NameValue[]
getNameValueArray ()
long getTimestampMillis ()
ErrorTrace getErrorTrace ()
ACSException getACSException ()
void log (Logger logger)

Appendix D. AcsJCompletion Public Methods

Public Member Functions *(See ACS API for full API)*

int getType ()
int getCode ()
long getTimeStamp ()
boolean isError ()
AcsJException getAcsJException ()
Completion toCorbaCompletion ()