CAPE-OPEN

Delivering the power of component software
and open standard interfaces
in Computer-Aided Process Engineering

Open Interface Specification:
Error Common Interface

www.colan.org

ARCHIVAL INFORMATION

Filename	Error Common Interface.doc
Authors	CO-LaN consortium
Status	Public
Date	August 2003
Version	version 7
Number of pages	53
Versioning	version 7, reviewed by Jean-Pierre Belaud, August 2003
	version 6, Methods & Tools group, July 2001
Additional material	
Web location	www.colan.org
Implementation specifications version	CAPE-OPENv1-0-0.idl (CORBA) CAPE-OPENv1-0-0.zip and CAPE-OPENv1-0-0.tlb (COM)
Comments	

IMPORTANT NOTICES

SUMMARY

This document founds the specification document for the error handling. A common strategy is defined in order to have a full uniform error management within the CAPE-OPEN specifications. The error handling is an important step for all development as well as for the definition of a good business standard such CAPE-OPEN. This strategy should be used by all CAPE-OPEN interfaces and therefore by all the resulting implementation.

For each business part (Thermo, Numr, Unit, …) the errors which could occur have to be carried out not only at the interface specification level within the IDL files but also at the analysis phase through the Use-Case and at the design phase through the diagrams. This document forms the basis of CAPE-OPEN error handling.

This specification document belongs to the family of *CAPE-OPEN Common Interfaces* such as Parameter and Identification.

ACKNOWLEDGEMENTS

CONTENTS

LIST OF FIGURES

1. Introduction

This report gives the guidelines to manage the error within any CO interface. So only the errors which can occur through the operations of a CO interface object are specified. Many other types of error exist such as the error coming from the middleware, the operating system, the installation program, the client application, … Their management are specific and proprietary. They are out of the scope of this document.

By definition the error is an abnormal termination. It represents a binary status; either there is no error or an error occurs. When a request is made, if this request is successful it raises no error otherwise it raises an error. When an error occurs, the execution is immediately aborted. Obviously the fact to decide if an error takes place belongs to the software developer. Additionally, he is responsible for notifying the user or caller that an error situation has occurred so that he can react properly. However the possible types of error are fixed by the CO specification (Open Interface Document) because all CO components are meant to be plug-and-play components with standardized error behaviour.

The error strategy is first defined from a conceptual view. Thus the strategy is independent from any architecture, system and implementation language. The result uses the UML notation. Then the error strategy is applied to the COM and CORBA architecture. As the both own some limited inner error management, some tunings are necessary in order to supply the specification (IDL files).

This document describes a classification and a hierarchy of potential errors occurring in the CO standard. These errors are common to all the CO interfaces which can easily reuse them. In order to bring some flexibility, the error design defines an error code. This code designates subcategory of error and its assignment of value is left to each component implementation. Additionally to the common errors described in this document, a CO interface can always define its specific own errors within its scope.

The implementation and validation of this strategy will be done within the GCO project where COM and CORBA component developments are planned.

An additional concept is related to the error handling; the warning concept. This concept is not defined here and is considered as a separate specification. In fact it is another CO common interface such as Parameter and Identification. It allows to "moderate" the binary running of error (success or not success). The error handling can be rigid in front of some circumstances. For example when a thermodynamic property computation is requested with data outside the bound limit, raising an error or not is not definitely the good way. The computation should be able to go on with an event detailing the warning. At present the CO standard does not specify the warning concept. As a minimum effort, the component software can display today its warnings through the reporting.

2. Requirements

This chapter introduces the requirements developed by the project team. It contains a textual description followed by some use cases. These requirements express the need for the CO standard to have a uniform and well designed error handling.

2.1 Textual requirements

Within this chapter additionally to the true requirements we give some general technical information and the current status of error handling in the CO standard. That will allow understanding the future design.

In this document in most cases, we use the term “error” when we deal with analysis and design phase and “exception” for the specification phase. The term “exception” is also used in order to refer to the inner mechanism of languages such as C++, Java and CORBA IDL.

2.1.1 Technical information

Error handling is a contentious issue among developers. For a long time, most C programmers subscribed to the every-function-call-returns-an-error-code style of coding. While effective, this error-handling mechanism has its drawbacks. To start with, error-handling code is sprinkled throughout an application, which makes the code hard to read and costly to maintain. This approach can also be inefficient because it requires a great deal of usually unnecessary code that simply checks for error return codes.

More recently, languages such as C++ and Java have introduced an error-handling mechanism known as exceptions. Exceptions are run-time errors by another name. Programming languages that support exceptions have special language directives that allow exceptions to be caught and raised. Raising an exception is a way to report an error; catching an exception means intercepting and responding to the error condition. C++ and Java support the try and catch keywords for trapping exceptions, while the “throw” keyword is used to raise exceptions. Even Microsoft Visual Basic supports trapping exceptions with the On Error statement; you use the Err.Raise method to raise exceptions. Additionally, in object oriented languages like C++ or Java exceptions are more or less treated like objects. This means that the developer can use inheritance mechanisms for defining errors. This facilitates the implementation of errors hierarchies which can be very convenient for error handling. As exceptions have similar traits compared to objects in OO languages they can be augmented with arbitrary information (e.g. strings, methods, numeric values) to give additional information about the error that occurred.

Generally two main families of exceptions are distinguished: the system (also called standard) exceptions and the user exceptions. The system exceptions are architecture dependant and a priori the implementation do not deal with them. Their handling and their design are specific to COM and CORBA. This document is interested in the user exceptions that is to say the exceptions within the CO interfaces.

From now except if explicitly notified, the term exception is related to the user exception.

THE UML CONCEPTUAL VIEW

One main aspect in the specification and documentation of an interface/class behaviour is located in the exception specification that the operations can cause. The exceptions should be explicitly designed.

In UML, the exceptions are a kind of signal (an event with a stimulus between instances) which are stereotype classes <<Exception>>. They can be added to the operation specification.

In order to design the exceptions, we need:

- To think on exceptional conditions which can come from each CO interface as well as each involved operations;
- To organise the exceptions into a hierarchy;
- To specify for each operation the exceptions that it can produce.

The first and second item is solved by this document while the third is under the responsibility of CO interface designers.

THE COM/COM+ VIEW

To the designers of COM, error handling represented a unique challenge because each programming language has its own ideas about error handling. As a language-neutral component architecture, COM must deal with errors in a way that is compatible with all major programming languages, some of which might not support exceptions. To meet that goal, COM supports both the errors-as-return-codes technique and the more modern idea of exceptions.

For method return codes, COM defines the HRESULT, a 32-bit value that identifies an error. In the exception-handling arena, COM defines several interfaces that support those semantics. Some high-level languages such as Visual Basic automatically map COM exceptions to the language's native exception-handling mechanism. In lower-level languages such as C++, COM exceptions are not automatically mapped to the C++ exception-handing mechanism. Instead, C++ developers must work directly with the COM exception interfaces or roll their own mapping layer that converts COM exceptions to C++ exceptions and vice versa. The Microsoft Java Virtual Machine (VM) also provides nearly automatic COM to Java exception translation. The error-handling technique you employ will depend on the programming language you use and your preferences about how to deal with errors.

In C++ COM any class can represent an exception as long as it supports the IErrorInfo interface. Additional interfaces can also be supported to allow access to extra information about an exception.

In VB it seems that the definition of the exception object is fixed. It might be possible to provide VB classes that implement IErrorInfo and other more specific interfaces as long as it is possible to call SetErrorInfo and GetErroInfo from VB.

It is not possible to define directly exceptions in MIDL.

THE CORBA VIEW

CORBA offers direct support for exceptions in a similar way Java or C++ do. Therefore, an exception in CORBA is an indication that an operation request was not performed successfully. An exception may be accompanied by additional, exception-specific information. The additional, exception-specific information is a specialised form of record. As a record, it may consist of any of the types. But an important conceptual difference in the CORBA exception concept compared to Java or C++ is that CORBA exceptions do not support inheritance. This means that an error hierarchy cannot be built directly on CORBA IDL language constructs.

Standard exceptions:

All signatures implicitly include the system exceptions; here are some details on the standard system exceptions in order to understand the approach applied in CORBA and then extract some technical aspects with the aim of our user exception definition.

A set of standard exceptions is defined corresponding to standard run-time errors which may occur during the execution of a request. This section presents the standard exceptions defined for the ORB. These exception identifiers may be returned as a result of any operation invocation, regardless of the interface specification. Standard exceptions may not be listed in raises expressions. In order to bound the complexity in handling the standard exceptions, the set of standard exceptions should be kept to a tractable size. This constraint forces the definition of equivalence classes of exceptions rather than enumerating many similar exceptions. For example, an operation invocation can fail at many different points due to the inability to allocate dynamic memory. Rather than enumerate several different exceptions corresponding to the different ways that memory allocation failure causes the exception (during marshalling, unmarshalling, in the client, in the object implementation, allocating network packets), a single exception corresponding to dynamic memory allocation failure is defined.

Each standard exception includes a minor code to designate the subcategory of the exception.

Each standard exception also includes a completion_status code which takes one of the values {COMPLETED_YES, COMPLETED_NO, COMPLETED_MAYBE}. These have the following meanings:

- COMPLETED_YES: the object implementation has completed processing prior to the exception being raised.
- COMPLETED_NO: the object implementation was never initiated prior to the exception being raised.
- COMPLETED_MAYBE: the status of implementation completion is indeterminate.

As illustration, UNKNOWN, BAD_PARAM, NO_MEMORY, COM_FAILURE, … are standard exceptions. Two particularly interesting standard exceptions could be the NO_IMPLEMENT and the NO_MEMORY exceptions. The first one indicates that a method call failed because the method implementation is missing, the second one means that a call was aborted due to a lack of memory. Both situations possibly could arise in the CO context and should therefore be taken into account in the error handling strategy.

User exceptions:

Exception declarations permit the declaration of struct-like data structures which may be returned to indicate that an exceptional condition has occurred during the performance of a request. The syntax is as follows: exception identifier { member(s) }

Each exception is characterised by its IDL identifier, an exception type identifier, and the type of the associated internal additional information (as specified by the <member> in its declaration). If an exception is returned as the outcome to a request, then the value of the exception identifier is accessible to the programmer for determining which particular exception was raised.

If an exception is declared with members, a programmer will be able to access the values of those members when an exception is raised. If no members are specified, no additional information is accessible when an exception is raised.

As it acts as a struct, neither operation nor attribute can be defined and obviously the inheritance is forbidden to the exceptions.

Language mappings:

Let us look what the CORBA exceptions become from an implementation view. Following the CORBA language mapping, the user exception defined in the IDL file are translated for Java and C++ in classes inherited from UserException class; org.omg.CORBA.UserEception for Java and CORBA::UserException for C++. The class CORBA::Exception is the base class.

Similarly the standard exceptions inherit from a SystemException class. Here below the class diagram:

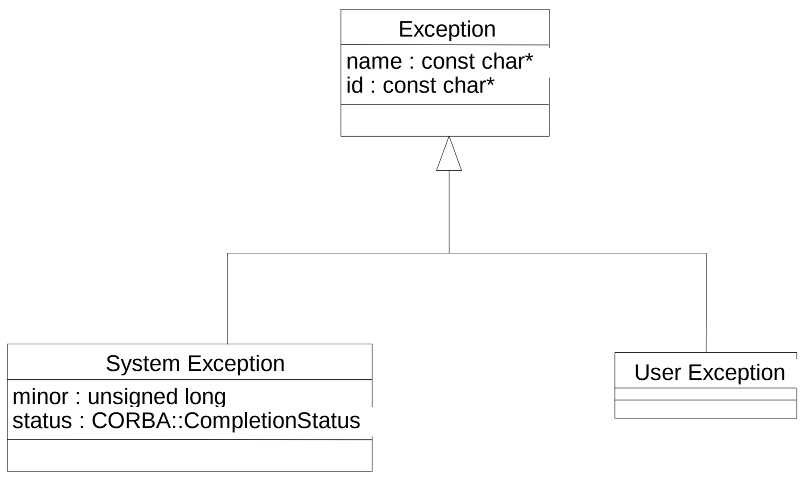

Figure 1 Exceptions diagram according to the C++ language mapping

Note that consequentially our C++/Java user exceptions will have for CORBA architecture two attributes: name and id due to the inheritance.

<u>JAVA AND C++ VIEW</u>

Obviously, the exception notion is present in these both languages. The exceptions are classes and so can have methods, attributes and can belong to an exception complete hierarchy.

However there are differences: any C++ class can be an exception while the Java class has to inherit from java.lang.Exception.

<u>CONCLUSION</u>

From an analysis and design view we have through UML a powerful way to design our error handling independently to middleware and implementation languages. From a specification view, COM and CORBA bring some limitations due to their inner mechanism. And it appears that COM and CORBA are more restrictive than the object-oriented languages.

2.1.2 The current status in the CO specification

<u>COM IDL</u>

In CAPE-OPENv0.9.tlb no definitions for error handling are included.

As information, the current implementations of the CO plugs used the following pattern when an error is encountered:

- If a CO interface method does not return an error code (or boolean). The plug sends a COM exception containing at least the fields: number, description and source.

- If a CO interface method has a boolean return value:

    - If an exception happens during the method implementation, the plug should catch it and return false (failing value).

    - If the method also has an out string argument, the plug will use it to describe the failure.

However, since the socket can never be sure that the plug will catch its own exceptions, the sockets are catching the exceptions raised by any CO plug.

CORBA IDL

In CAPE-OPENv0.9.1.idl, there was no error handling (no exception) since the CORBA standard designers are waiting for this document. Before the distribution of the library CAPE-OPENv0.9.1.idl, work took place and several different strategies were suggested.

As information, the former implementations of CO interfaces used the CORBA inner mechanism of exception pattern. Each CO interface defined its own exception types, exception body and strategy to be applied.

2.1.3 Desirable characteristics

Here is the summary of the desired characteristics for the CO error handling:

- We go for a uniform approach that is consistent in all CO specifications. Moreover, we aim at defining the same types of errors for COM and CORBA although they cannot be handled in the same way on technical level because there is no real error mechanism in COM. But on a conceptual level the behaviour of a component in an error state is clearly defined regardless of the middleware used;

- We start to define error conditions from a conceptual view using the UML notation and then see how we can translate this to COM and CORBA error handling mechanisms;

- The strategy must propose common errors which are vertical to all the components (Thrm, Unit , Numr, …). These errors are common to many components. Although, this document will include definitions of error conditions that are only dedicated to certain interfaces additionally to the more general errors.

- These common error types could then be refined in the different scopes of interfaces according to the error conditions that can occur in a certain context. Thus each CO interface designer has to be able to specify specific errors;

- Therefore there are two kinds of CO errors: the common errors and the specific errors. The common errors can be reused by the CO interface designer directly as a CO common interface. The specific errors are dedicated to a particular scope of an interface. Even if this document gives some information on how to design the specific errors, it is only focus on the common errors.

- As advised by the UML guide and in fact basically by the object-oriented paradigm, the common errors should form the basis of an error hierarchy. The top of the hierarchy is similar to the CORBA view: a user error inheriting from a root error;

- Some errors, at the top of the hierarchy, are generic errors. They are abstract errors and can not be instantiated. But most of errors are can be instantiated by any CO component.

- We have not a unique common error with a field giving the type of the appropriate error. To do that would be very convenient from a point of CO interface designers view since all CO operations would specify one general and single error (that would result in an easy design !). But we would have lost partially the exception system capability : in order to catch a precise error type, the CO component developer would be constraint to code a condition block within this general error rather than using the inner mechanism of languages. Additionally (at least on the CORBA side), the developer can see on IDL level which user exceptions may be caused by a call to a certain operation as every exception possibly thrown must be declared. This contract on the potential errors is a proof of a better design;

- For each operation, we should find a compromise between the approach which defines only few errors and the approach which defines many errors. If an operation specifies many errors, the developer work becomes more intense. We try to define only the errors which can really occur;

- To keep a degree of freedom for component developers, a code is included in the body of the error. The value and the meaning of this code is implementation dependent;

- Each CO operation specifies an error called Unknown. That allows the developer to outcome an error which is not specified by the interface specification. Naturally the Unknown error should be used only if any other error can not be applied.

- Each CO operation specifies an error called Invalid Argument except the ones which do not deal with any data. That means that if the operation has at least one argument the error Invalid Argument has to be specified.

- There is no different meanings of an error. That means that the error concept should not be used in order to provide a warning, an information message or something else. The CO error handling is only focussed on failure message. By definition, the error is an abnormal termination;

2.2 Use cases

This section and the following present the requirements in a more formal way using UML models: Use Cases, Use Case Maps and Sequence Diagrams.

The use cases for the general error handling concepts will make no references to specific interfaces as they apply to all CO interfaces in the same way. These use cases are termed general use cases. The use cases for more specific errors will reference some interfaces to which the particular use case may be connected. These uses cases will be termed specific use cases.

2.2.1 Actors

- **Caller.** Any person or software component that issues a call to a method of an arbitrary CO software component. Therefore, almost everything and everyone in the CO context may be a caller. As concrete examples under the CO scope some possible callers:

  - **Physical Properties Developer.** The human being who is notionally a physical properties expert and will set up the physical property options for use by the normal simulator end user. In principle, the Physical Properties developer is responsible for physical properties quality assurance in the organisation using simulation.

  - **Unit Operation Component.** The unit operation will make method calls to a material object in order to retrieve or store data.

  - **Material Object Component.** The material object will make calls to a thermo properties package in order to start some thermodynamic computation or to retrieve some thermodynamical data.

  - **Flowsheet Builder.** The person who sets up the flowsheet, the structure of the flowsheet, chooses thermo models and the unit operation models that are in the flowsheet. This person hands over a working flowsheet to the Flowsheet User. The Flowsheet Builder can act as a Flowsheet User.

- **Flowsheet User.** The person who uses an existing flowsheet. This person will put new data into the flowsheet, rather than change the structure of the flowsheet.

2.2.2 List of Use Cases

- UC-001: General Failed Method Computation
- UC-002: General Failed Method Invocation
- UC-003: General Error Determination
- UC-004: General CO Error Handling
- UC-005: Specific Unknown Parameter Request
- UC-006: Specific Unsupported Parameter Request
- UC-007: Specific Invalid Parameter Passed
- UC-008: Specific Invalid Object Reference Passed
- UC-009: General Unsupported Method Called
- UC-010: General Computational Failure
- UC-011: General Storing Failure
- UC-012: General Restoring Failure

2.2.3 Use Cases Maps

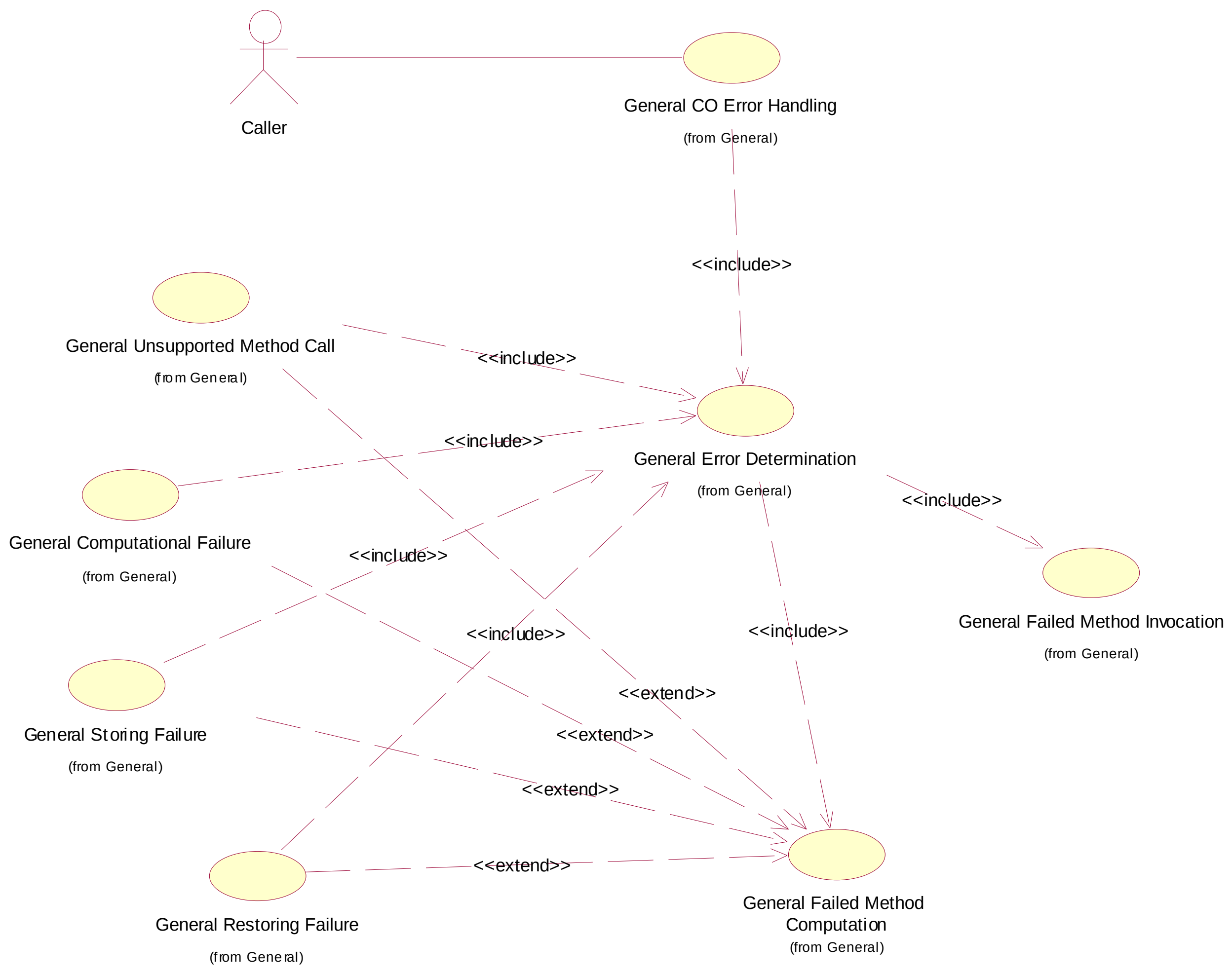

Figure 2 The use-cases map

2.2.4 Use Cases

This subsection lists all the Use Cases.

UC-001: NERAL FAILED METHOD COMPUTATION

Actors: Caller Priority: High Classification: General Context: A piece of software or a user (caller) is performing a calculation which involves code that is located in an external component. Pre-conditions: The component that should be called is available in the system. Flow of events: The caller invokes the components method. The component performs the computation and an error situation occurs. The component creates an error.

Post-conditions: The component still exists in the system.

Errors:

Uses:

Extends:

UC-002: General Failed Method Invocation

Actors: Caller

Priority: High

Classification: General

Context: A piece of software or a user (caller) is performing a calculation which involves code that is located in an external component.

Pre-conditions: The component is not available.

Flow of events: The caller invokes the components method. The component is not available which is discovered by the middleware platform. The middleware system creates a standard error. This error is managed by the caller using platform dependant system.

Post-conditions:

Errors:

Uses:

Extends:

UC-003: General Error Determination

Actors: Caller

Priority: High

Classification: General

Context: The caller tried to perform a computation in an external component which has failed for some reason.

Pre-conditions: The method invocation has created an error which was passed to the caller.

Flow of events: The caller analyses the error and tries to determine the reason for the error. Additionally, the caller tries to analyse when the errors was created and how the status of the external computation looks like (complete, incomplete, unknown,...).

Post-conditions: The information what error has occurred is known and stored.

Errors:

Uses: UC-002 (General Failed Method Invocation.), UC-001 (General Failed Method Computation)

Extends:

UC-004: General CO Error Handling

Actors: Caller

Priority: High

Classification: General

Context: The caller tried to perform a computation in an external component which has failed for some reason. The component has created a CO error which was received and analysed by the caller.

Pre-conditions: The caller knows what kind of error has occurred.

Flow of events: The caller tries to handle the CO error in some way according to its kind.

Post-conditions:

Errors:

Uses: UC-003 (General Error Determination)

Extends:

UC-005: Specific Unknown Parameter Request

Actors: Material Object Component

Priority: High

Classification: Specific

Context: The Material Object tries to compute or request a property by calling a Properties Package method.

Pre-conditions: The Property Package component is alive.

Flow of events: The MO calls a PP's method and passes a property (as string) which is not known within the CO standard. Then the PP creates an error which indicates this situation and passes it to the caller.

Post-conditions: The PP is still alive. The MO knows why the method call failed.

Errors:

Uses: UC-003 (General Error Determination)

Extends: UC-001 (General Failed Method Computation)

UC-006: Specific Unsupported Parameter Request

Actors: Material Object Component

Priority: High

Classification: Specific

Context: The Material Object tries to compute or request a property by calling a Properties Package method.

Pre-conditions: The Property Package component is alive.

Flow of events: The MO calls a PP's method and passes a property (as string) which is a valid CO property description but is not supported by the PP. Then the PP creates an error which indicates this situation and passes it to the caller.

Post-conditions: The PP is still alive. The MO knows why the method call failed.

Errors:

Uses: UC-003 (General Error Determination)

Extends: UC-001 (General Failed Method Computation)

UC-007: SPECIFIC INVALID PARAMETER PASSED

Actors: Material Object Component

Priority: High

Classification: Specific

Context: The Material Object tries to compute or request a property by calling a Properties Package method.

Pre-conditions: The Property Package component is alive.

Flow of events: The MO calls a PP's method and passes a parameter which has an invalid value meaning that either this parameter is out of bounds or has an unspecified NULL value. Then the PP creates an error which indicates this situation and passes it to the caller.

Post-conditions: The PP is still alive. The MO knows why the method call failed.

Errors:

Uses: UC-003 (General Error Determination)

Extends: UC-001 (General Failed Method Computation)

UC-008: SPECIFIC INVALID OBJECT REFERENCE PASSED

Actors: Caller

Priority: High

Classification: Specific

Context: The caller tries to perform an external computation and has to pass a component reference as parameter.

Pre-conditions: The external component is alive. The parameter reference is invalid for some reason.

Flow of events: The caller calls the external method and passes the object reference. During the external computation the external component recognises that the parameter reference is invalid so that the parameter component cannot be called. The external component creates an appropriate error.

Post-conditions: The external component is still alive. The caller knows why the method call failed.

Errors:

Uses: UC-003 (General Error Determination)

Extends: UC-001 (General Failed Method Computation)

UC-009 GENERAL UNSUPPORTED METHOD CALLED

Actors: Caller

Priority: High

Classification: General

Context: The caller tries to perform an external computation by calling a method.

Pre-conditions: The external component is alive. The external component does not support this method.

Flow of events: The caller calls the external method. Because the external component does not support (i.e. has not implemented) this method it creates an appropriate error.

Post-conditions: The external component is still alive. The caller knows why the method call failed.

Errors:

Uses: UC-003 (General Error Determination)

Extends: UC-001 (General Failed Method Computation)

UC-010: General Computational Failure

Actors: Caller

Priority: High

Classification: General

Context: The caller tries to perform an external computation by calling a method.

Pre-conditions: The external component is alive. All parameters passed are ok.

Flow of events: The caller calls the external method. The external computation is started but fails to complete for some reason (out of memory, non termination within a certain time, algorithm specific errors,...). The external component throws an appropriate error.

Post-conditions: The external component is still alive. The caller knows why the method call failed.

Errors:

Uses: UC-003 (General Error Determination)

Extends: UC-001 (General Failed Method Computation)

UC-011: General Storing Failure

Actors: Caller

Priority: High

Classification: General

Context: The caller tries to make an external component persistent.

Pre-conditions: The external component is alive.

Flow of events: The caller calls the external persistence method. The storing of the external component fails for some reason (no space, insufficient access rights, component cannot be made persistent). The external component throws an appropriate error.

Post-conditions: The external component is still alive. The caller knows why the method call failed.

Errors:

Uses: UC-003 (General Error Determination)

Extends: UC-001 (General Failed Method Computation)

UC-012: GENERAL RESTORING FAILURE

Actors: Caller

Priority: High

Classification: General

Context: The caller tries to restore an external component which was made persistent before.

Pre-conditions:

Flow of events: The caller calls a restore method. The restore operation fails and an appropriate error is thrown.

Post-conditions:

Errors:

Uses: UC-003 (General Error Determination)

Extends: UC-001 (General Failed Method Computation)

2.3 Sequence diagrams

Two sequence diagrams are showed. They describe the interaction between the Caller and any CO component when a CO error occurs during a request of a CO operation.

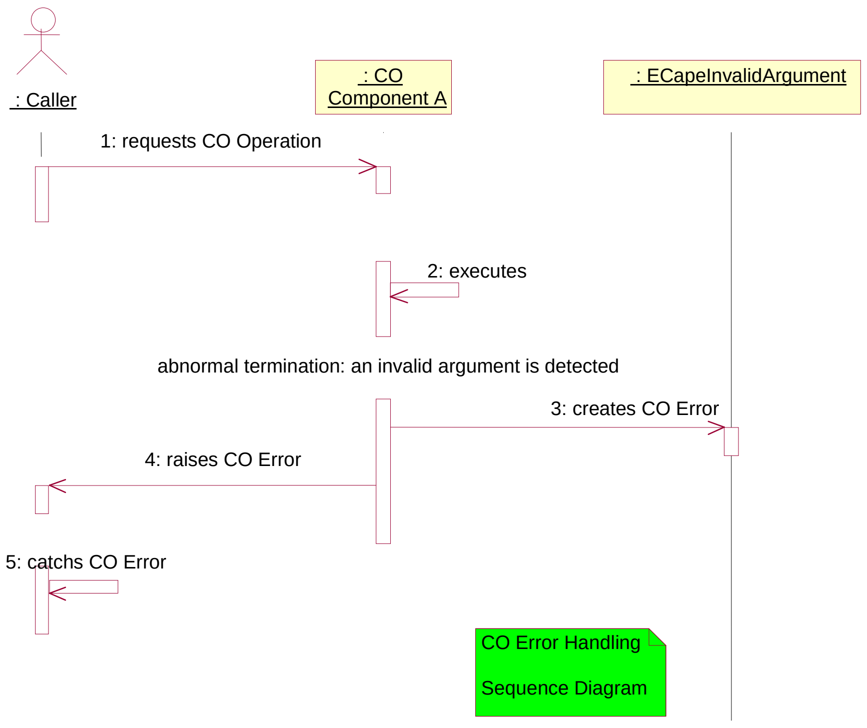

Figure 3 First Sequence diagram for an invalid argument error

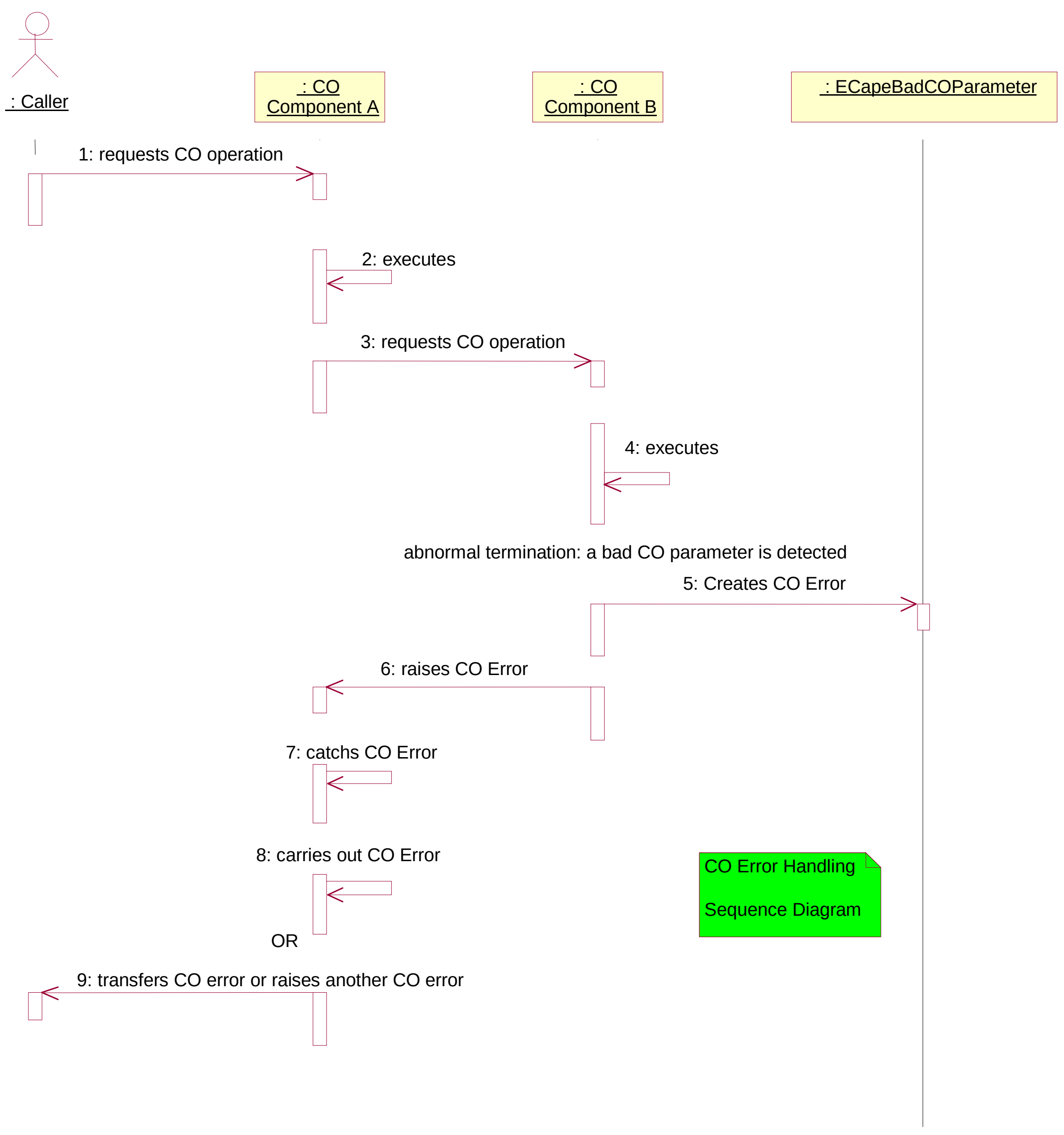

Figure 4 Second sequence diagram for an invalid argument error

3. Analysis and Design

This chapter introduces the design. It contains a textual description followed by UML diagrams.

3.1 Overview

Even if formally speaking the design depends on the view, the Error Handling strategy for CO standard requests an agreement on the design from the conceptual view. Then the translation into COM and CORBA will follow technical rules the M&T group has defined.

This section is only focus on the design from the conceptual view.

- An error type is represented by a class. The errors hierarchy owns a base class called ECapeRoot placed in the Root package. The two other packages System and CapeUser depends on the Root package.
- The System package contains the system (also called standard) errors. As its content is linked to the platform, its design does not need to be defined.
- The CapeUser package has all the errors of CAPE-OPEN users. The error ECapeUser is the base class of the CapeUser package. It defines the minimum state of all the CO errors.
- The classification identifies four sources of errors: the data, the implementation, the computation and the persistence.
- All the errors have information fields as attributes which document the error an its origin. The attributes which are mandatory are name, interface and operation. That means that the CO component developer has to fill these fields.
- There are four abstract errors and more than 20 real errors.

3.2 Error Class diagrams

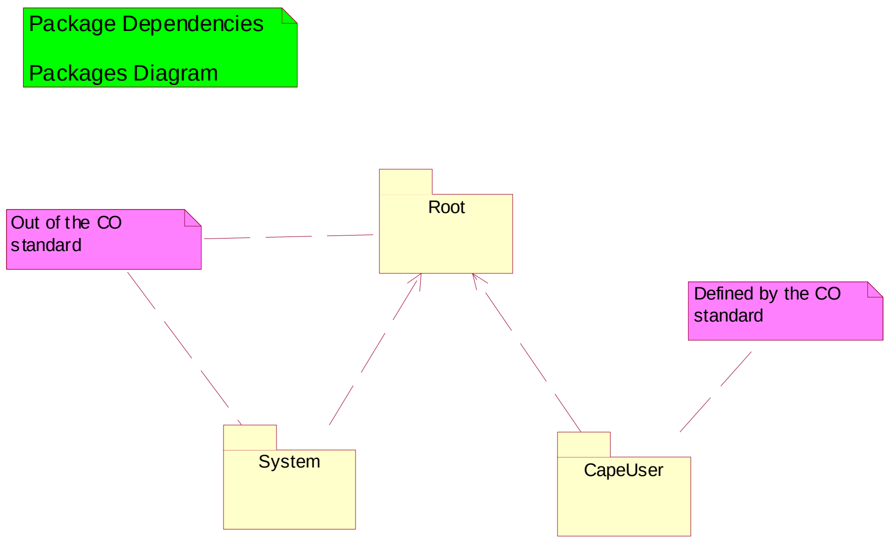

Figure 5 The package dependencies diagram

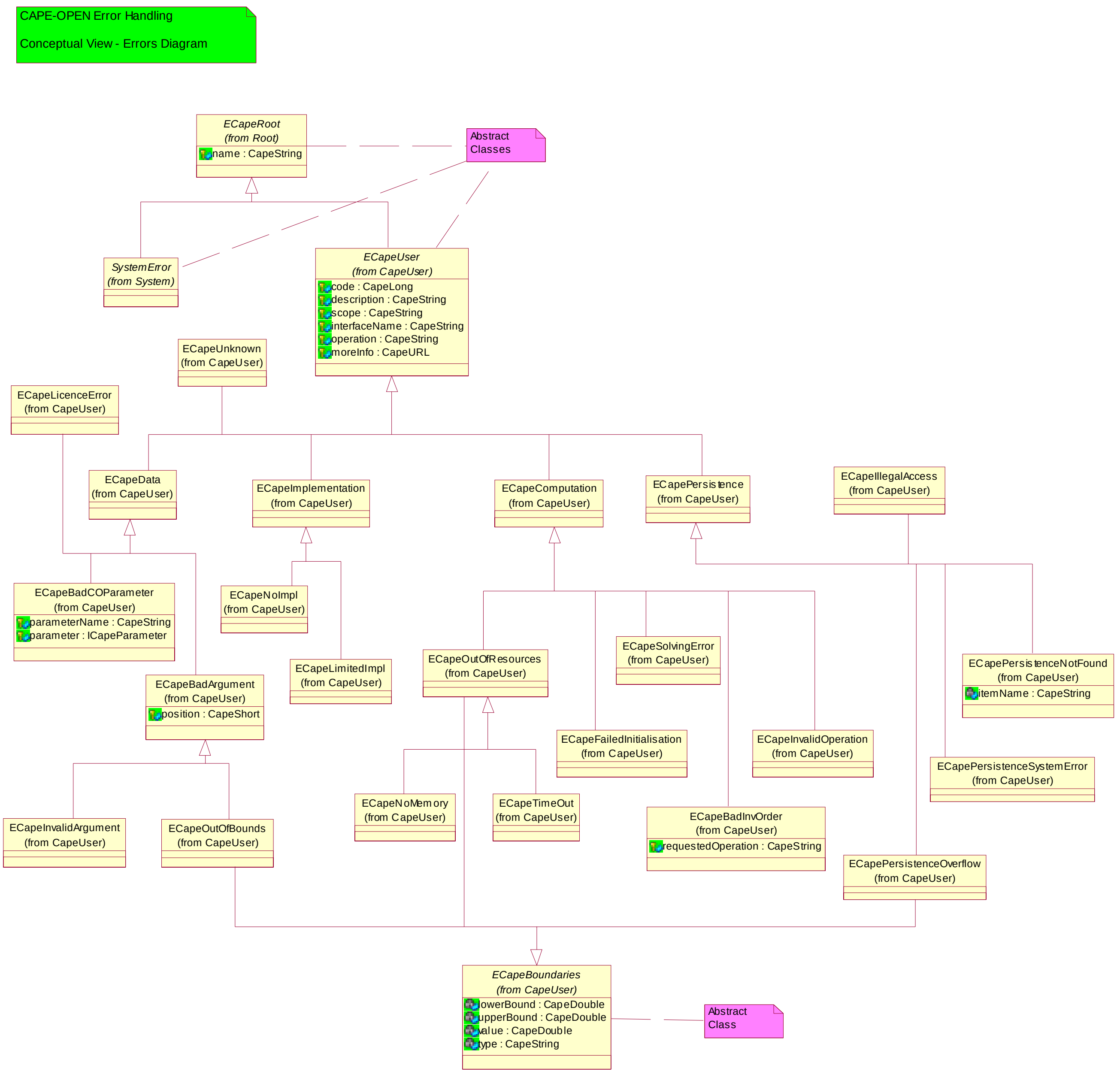
CAPE-OPEN Error Handling
Conceptual View - Errors Diagram
ECapeRoot (from Root)
name : CapeString
Abstract Classes
SystemError (from System)
ECapeUser (from CapeUser)
code : CapeLong
description : CapeString
scope : CapeString
interfaceName : CapeString
operation : CapeString
moreInfo : CapeURL
ECapeUnknown (from CapeUser)
ECapeLicenceError (from CapeUser)
ECapeData (from CapeUser)
ECapeImplementation (from CapeUser)
ECapeComputation (from CapeUser)
ECapePersistence (from CapeUser)
ECapeIllegalAccess (from CapeUser)
ECapeBadCOParameter (from CapeUser)
parameterName : CapeString
parameter : ICapeParameter
ECapeNoImpl (from CapeUser)
ECapeLimitedImpl (from CapeUser)
ECapeOutOfResources (from CapeUser)
ECapeSolvingError (from CapeUser)
ECapePersistenceNotFound (from CapeUser)
itemName : CapeString
ECapeBadArgument (from CapeUser)
position : CapeShort
ECapeFailedInitialisation (from CapeUser)
ECapeInvalidOperation (from CapeUser)
ECapePersistenceSystemError (from CapeUser)
ECapeInvalidArgument (from CapeUser)
ECapeOutOfBounds (from CapeUser)
ECapeNoMemory (from CapeUser)
ECapeTimeOut (from CapeUser)
ECapeBadInvOrder (from CapeUser)
requestedOperation : CapeString
ECapePersistenceOverflow (from CapeUser)
ECapeBoundaries (from CapeUser)
lowerBound : CapeDouble
upperBound : CapeDouble
value : CapeDouble
type : CapeString
Abstract Class

Figure 6 The error class diagram

3.3 Errors descriptions

Each error (class) is presented together with its corresponding attributes in regards to the conceptual view:

3.3.1 ECapeRoot

Error Name	ECapeRoot

Description

The base class of the errors hierarchy.

The System package and the CapeUser package depend on this error.

This is an abstract class. No real error can be raised from this class.

Attributes

Name	Type	Description
name	CapeString	A short description of the error. This is a mandatory field.

3.3.2 ECapeUser

Error Name	ECapeUser

Description

The base class of the CO errors hierarchy. It defines the minimum state of a CO error.

This is an abstract class. No real error can be raised from this class.

Attributes

Name	Type	Description
code	CapeLong	Code to designate the subcategory of the error. The assignment of values is left to each implementation. So that is a proprietary code specific to the CO component provider.
description	CapeString	The description of the error.
scope	CapeString	The scope of the error. The list of packages where the error occurs separated by "::". For example CapeOpen::Common::Identification.
interfaceName	CapeString	The name of the interface where the error is thrown. This is a mandatory field.
operation	CapeString	The name of the operation where the error is thrown. This is a mandatory field.
moreInfo	CapeURL	An URL to a page, document, web site, … where more information on the error can be found. The content of this information is obviously implementation dependent.

3.3.3 ECapeBoundaries

Error Name	ECapeBoundaries

Description

A "utility" class which factorises a state which describes the value, its type and its boundaries.

This is an abstract class. No real error can be raised from this class.

Attributes

Name	Type	Description
lowerBound	CapeDouble	The value of the lower bound.
upperBound	CapeDouble	The value of the upper bound.
value	CapeDouble	The current value which has led to an error.
type	CapeString	The type/nature of the value. The value could represent a thermodynamic property, a number of tables in a database, a quantity of memory, …

3.3.4 ECapeUnknown

Error Name	ECapeUnknown

Description

The error to be raised when other error(s), specified by the operation, do not suit.

Attributes

None

3.3.5 ECapeData

Error Name	ECapeData

Description

The base class of the errors hierarchy related to any data. The data are the arguments of operations, the parameters coming from the Parameter Common Interface and information on licence key.

Attributes

None

3.3.6 ECapeLicenceError

Error Name	ECapeLicenceError

Description

An operation can not be completed because the licence agreement is not respected.

Of course, this type of error could also appear outside the CO scope. In this case, the error does not belong to the CO error handling. It is specific to the platform.

Attributes

None

3.3.7 ECapeBadCOParameter

Error Name	ECapeBadCOParameter

Description

A parameter, which is an object from the Parameter Common Interface, has an invalid status.

Attributes

Name	Type	Description
parameterName	CapeString	The name of the CO parameter
parameter	IcapeParameter	The parameter

3.3.8 ECapeBadArgument

Error Name	ECapeBadArgument

Description

An argument value of the operation is not correct.

Attributes

Name	Type	Description
position	CapeShort	The position of the argument value within the signature of the operation. First argument is as position 1.

3.3.9 ECapeInvalidArgument

Error Name	ECapeInvalidArgument

Description

An invalid argument value was passed.

For instance the passed name of the phase does not belong to the CO Phase List.

Attributes

None

3.3.10 ECapeOutOfBounds

Error Name	ECapeOutOfBounds

Description

An argument value is outside of the bounds.

Attributes

None

3.3.11 ECapeImplementation

Error Name	ECapeImplementation

Description

The base class of the errors hierarchy related to the current implementation.

Attributes

None

3.3.12 ECapeNoImpl

Error Name	ECapeNoImpl

Description

The operation is “not” implemented even if this operation can be called due to the compatibility with the CO standard. That is to say that the operation exists but it is not supported by the current implementation.

Attributes

None

3.3.13 ECapeLimitedImpl

Error Name	ECapeLimitedImpl

Description

The limit of the implementation has been violated.

An operation may be partially implemented for example a Property Package could implement TP flash but not PH flash. If a caller requests for a PH flash, then this error indicates that some flash calculations are supported but not the requested one.

The factory can only create one instance (because the component is an evaluation copy), when the caller requests for a second creation this error shows that this implementation is limited.

Attributes

None

3.3.14 ECapeComputation

Error Name	ECapeComputation

Description

The base class of the errors hierarchy related to the calculation.

Attributes

None

3.3.15 ECapeOutOfResources

Error Name	ECapeOutOfResources

Description

The physical resources necessary to the execution of the operation are out of limits.

Attributes

None

3.3.16 ECapeNoMemory

Error Name	ECapeNoMemory

Description

The physical memory necessary to the execution of the operation is out of limit.

Attributes

None

3.3.17 ECapeTimeOut

Error Name	ECapeTimeOut

Description

The time-out criterion is reached.

Attributes

None

3.3.18 ECapeFailedInitialisation

Error Name	ECapeFailedInitialisation

Description

The pre-requisites are not valid. The necessary initialisation has not been performed or has failed.

Attributes

None

3.3.19 ECapeSolvingError

Error Name	ECapeSolvingError

Description

A numerical algorithm fails for any reasons.

Attributes

None

3.3.20 ECapeBadInvOrder

Error Name	ECapeBadInvOrder

Description

The necessary pre-requisite operation has not been called prior to the operation request.

Attributes

Name	Type	Description
requestedOperation	CapeString	The necessary prerequisite operation.

3.3.21 ECapeInvalidOperation

Error Name	ECapeInvalidOperation

Description

This operation is not valid in the current context.

Attributes

None

3.3.22 ECapePersistence

Error Name	ECapePersistence

Description

The base class of the errors hierarchy related to the persistence.

Attributes

None

3.3.23 ECapePersistenceOverflow

Error Name	ECapePersistenceOverflow

Description

There is an overflow of internal persistence system.

Attributes

None

3.3.24 ECapeIllegalAccess

Error Name	ECapeIllegalAccess

Description

The access to something within the persistence system is not authorised.

Attributes

None

3.3.25 ECapePersistenceNotFound

Error Name	ECapePersistenceNotFound

Description

The requested object, table, or something else within the persistence system was not found.

Attributes

Name	Type	Description

itemName	CapeString	The name of the item which has not been found.

3.3.26 ECapePersistenceSystemError

Error Name	ECapePersistenceSystemError

Description

There is a severe error within the persistence system.

Attributes

None

4. Interface Specifications

This section contains the COM and CORBA IDL instructions specific to the error handling.

To understand how these files have been generated from the Analysis and Design, go through the section Notes on the interface specifications.

4.1 COM IDL

```
// You can get these intructions in Error.idl file from CAPE-OPENv1-0-0.zip
```

4.2 CORBA IDL

```
// You can get these intructions in CAPE-OPENv1-0-0.idl within the
CAPEOPEN100::Common::Error module
```

5. Notes on the interface specifications

The purpose of these notes is to record the rationale for the decisions made in designing the interfaces and attributes.

5.1 Translation from conceptual view to COM

5.1.1 Technical information

As described in the technical information chapter of the requirements section, our design will be constraint by high level languages such as VB.

Although COM has a special return Code for expressing “No Implementation”: E_NOTIMPL = 0x80004001, the CAPE-OPEN components should use the following interface ECapeNOImpl instead of E_NOIMPL.

The COM standard states that, when a component raises an error, it must call SetErrorInfo, passing a pointer to an IErrorInfo interface(the standard COM error interface). The caller of the component should retrieve the pointer through function GetErrorInfo. Unfortunately, since the IErrorInfo interface does not have any attribute of type IDispatch, there is no way to use it to pass a reference to a ECapeRoot interface to the caller.

So, a possible COM implementation could be achieved with the following steps:

- The component creates an object which implements both IErrorInfo and an interface derived from ECapeUser.
- The component uses SetErrorInfo to pass a IDispatch pointer to the created object.
- The caller gets the IDispatch pointer through SetErrorInfo
- Now the caller has to learn the exact type of the CO exception that has been raised (eg. ECapeComputation, ECapeNoMemory, ...) Since COM calls always return a long value (called HRESULT), we will define a table that will map each CO error interface to a valid user HRESULT value. See below.
- The caller uses QueryInterface to cast the IDispatch pointer to the particular CO error interface specified by the HRESULT code.

VB PROBLEM

Unfortunately, VB does not support using the IErrorInfo interface, because exceptions are hidden by object “Err”. However, from VB we could call SetErrorInfo passing an IDispatch pointer to a ECapeRoot which would not support IErrorInfo. This causes two problems:

- We are breaking COM rules, since system COM exceptions will pass IErrorInfo pointers and CAPE-OPEN COM exception would pass pointers which don’t support this interface.
- Although we can call SetErrorInfo from VB, with this language we can only set the HRESULT return code calling Err.raise. And calling this method will always overwrite the pointer passed by SetErrorInfo

It might seem that the VB class ErrObject is the VB way to implement the IErrorInfo. However:

- Since VB does not allow inheritance, you cannot inherit IErrorInfo through it.

- Adding “Implements ErrObject” to a class module does not work.

- “dim errObj as ErrObject = New ErrObject” does not work either

VB SOLUTION

The only alternative is forcing that the component object is the component that implements the CAPE-OPEN error interfaces. This would bypass the COM Set/GetErrorInfo methods.

5.1.2 Resulting design

This design is specific to COM system. It results from the technical rules detailed in the previous section. As COM IDL does not support defined exceptions, they must be defined as plain COM interface.

In our conceptual view we use exceptions inheritance. COM does not support inheritance in the interface definition. Instead, if in the conceptual view interface C inherits from interface B, and the latter inherits from interface A, all these interfaces will be defined in the IDL file independently. The inheritance appears in the implementation of the classes, where implementing interface C obliges to also implementing interfaces A and B.

- All components that support the Error Common Interface must implement themselves all the error interfaces (those derived from ECapeUser) that the components are able to raise. At the same time, this feature provides a simple way to obtain the list of Error CO supported by a CAPE-OPEN class.

- When an operation fails, the component will return the standard COM HRESULT value. VB will pass it through the “number” property of the “Err” object, and the other languages will just return this error code (a 0 is returned when no error occurred). See below a table that maps each CO error interface to a valid user HRESULT value. So, if exception ECapeUnknown must be raised, the operation must return number OLE_E_FIRST + 1.

- The caller casts the called component’s pointer to the exact interface (lets call it I) obtained by checking the returned HRESULT error code in the table below. Actually, the caller is allowed to cast the pointer to any of the base interfaces of interface I (as defined in the conceptual view). Obviously, if an interface has no properties nor methods, it’s useless to get a pointer to that interface. That means that the CAPE-OPEN components are not obliged to implement these interfaces.

COM defines the first valid HRESULT value for user errors as 0x0040000 (defined as vbObjectError in VB and OLE_E_FIRST in C++). However, according to section "Strategies for Handling Errors in COM+", extracted from the MS Platform SDK help:

```
Use the FACILITY_ITF range of errors to report interface-specific errors. Interface-
specific errors should be in the FACILITY_ITF range of errors, between 0x0200 and 0xFFFF.
You can define a custom error code in Visual Basic as an offset from vbObjectError. Use
the MAKE_HRESULT macro in C++ to introduce an interface-specific error code, as shown in
the following example:
const HRESULT ERROR_NUMBER = MAKE_HRESULT (SEVERITY_ERROR, FACILITY_ITF, 200);
Unfortunately, if you try a VB client catching CAPE_OPEN exceptions, you'll see that COM
interpretes our codes and provides some confusing description.
You can read about the topic in
http://groups.google.com/groups?hl=en&safe=off&ic=1&th=983a0cd06aba4ea0,7&seekm=OZiCl%23c
u%23GA.176%40cppssbbsa02.microsoft.com
So, M&T decided to take the advise described in http://devguy.com/fp/Tips/COM/ and start
the CAPE-OPEN interface-specific errors at 500 instead of 200.

Executing MAKE_HRESULT macro, you obtain that the first available value is 0x80040500.
From now on, we will call it FIRST_E_INTERFACE_HR. Since 0xFFFF is the maximum value for
the FACILITY_ITF facility, the offset on FIRST_E_INTERFACE_HR must be between 1 and 64255
(0xFFFF-0x0500). We reserve the 0 offset for future uses.
```

In the COM IDL we will also define

```
// last HR value used for a CO error interface
const CapeErrorInterfaceHR LAST_USED_E_INTERFACE_HR = 0x80040517;
// hightes HR value that could be used to represent a CO error interface
const CapeErrorInterfaceHR LAST_E_INTERFACE_HR = 0x8004FFFF;
```

All operations of any CO components must always return a value from the table below. For those error interfaces that will be defined by new CO specifications, their designers must request a new value to the M&T group. We have to keep in mind that the maximum possible value will be `FIRST_E_INTERFACE_HR +` 64255.

CO Error interface	HRESULT value
ECapeRoot, ECapeUser, ECapeBoundaries	Since these are abstract errors, no components can raise them directly.
ECapeUnknown	FIRST_E_INTERFACE_HR + 1
ECapeData	FIRST_E_INTERFACE_HR + 2
ECapeLicenceError	FIRST_E_INTERFACE_HR + 3
ECapeBadCOParameter	FIRST_E_INTERFACE_HR + 4
ECapeBadArgument	FIRST_E_INTERFACE_HR + 5
ECapeInvalidArgument	FIRST_E_INTERFACE_HR + 6
ECapeOutOfBounds	FIRST_E_INTERFACE_HR + 7
ECapeImplementation	FIRST_E_INTERFACE_HR + 8
ECapeNoImpl	FIRST_E_INTERFACE_HR + 9
ECapeLimitedImpl	FIRST_E_INTERFACE_HR + 10
ECapeComputation	FIRST_E_INTERFACE_HR + 11
ECapeOutOfResources	FIRST_E_INTERFACE_HR + 12
ECapeNoMemory	FIRST_E_INTERFACE_HR + 13
ECapeTimeOut	FIRST_E_INTERFACE_HR + 14
ECapeFailedInitialisation	FIRST_E_INTERFACE_HR + 15
ECapeSolvingError	FIRST_E_INTERFACE_HR + 16
ECapeBadInvOrder	FIRST_E_INTERFACE_HR + 17
ECapeInvalidOperation	FIRST_E_INTERFACE_HR + 18
ECapePersistence	FIRST_E_INTERFACE_HR + 19
ECapeIllegalAccess	FIRST_E_INTERFACE_HR + 20
ECapePersistenceNotFound	FIRST_E_INTERFACE_HR + 21
ECapePersistenceSystemError	FIRST_E_INTERFACE_HR + 22
ECapePersistenceOverflow	FIRST_E_INTERFACE_HR + 23

5.2 Translation from conceptual view to CORBA

5.2.1 Technical information

As introduced in the Technical information section, due to the CORBA error inner mechanism some technical rules are decided in order to translate the errors from the conceptual view to the CORBA system. CORBA uses the exception mechanism.

CORBA offers direct support for exceptions using the IDL type "exception". An exception may be accompanied by additional, exception-specific information. The additional, exception-specific information is a specialised form of record. As a record, it may consist of any of the types. So an important conceptual difference in the CORBA exception is that CORBA exceptions do not support inheritance. This means that the error hierarchy defined in the section 3.2 can not be translated directly to the CORBA class diagram and to the CORBA IDL language.

Here are the technical directives that the M&T group has selected, that guarantees the compatibility between the analysis and the design of the conceptual view and the resulting design and specification of the CORBA view.

- The abstract errors (ECapeRoot, ECapeUser and ECapeBoundaries) are ignored. Each other error is translated to a CORBA exception. Its body needs to include the state of all the parent errors (abstract or not) according to the inheritance scheme. There are 23 exceptions.

- Following the language mappings (see The CORBA view section), all the exception classes (in C++ for example) inherit from the UserException class. This latter class has two public attributes: name as a string and id as a string. (in fact they are due to the inheritance from the base class Exception). Therefore, the attribute name present in the ECapeRoot is implicit. And the attribute name does not need to be included in the body of a CORBA exception.

- From an implementation point of view, any caller can try and catch any exception or the base exception. This base exception depends on the implementation language. For instance org.omg.CORBA.UserException for Java development and CORBA::UserException for C++ development.

- All the CO exceptions belong to a module called Error. As the Error Handling Strategy belongs to the family of the CO Common Interfaces, the Error module is put inside the module Common. The complete path is therefore CapeOpen::Common::Error.

- As the hierarchy following the inheritance scheme is not applicable, the CORBA module concept could be used. The exceptions could be classed with respect to the modules Data (corresponding to the ECapeData hierarchy) , Implementation (corresponding to the ECapeImplementation hierarchy), Computation (corresponding to the ECapeComputation hierarchy) and Persistence (corresponding to the ECapePersistence hierarchy). However this solution is not selected since the path to an exception would become too long (CapeOpen::Common::Error::Computation for example). So instead of using the CORBA modules, the packages Data, Implementation, Computation and Persistence are used. Comment lines reminds this classification.

- Any operation which specifies exception(s) has to use the path Common::Error::ECapeXxxx. At least any CO operation can raise the ECapeUnknown exception.

5.2.2 Resulting design

This design is specific to CORBA system. It results from the technical rules detailed in the previous section.

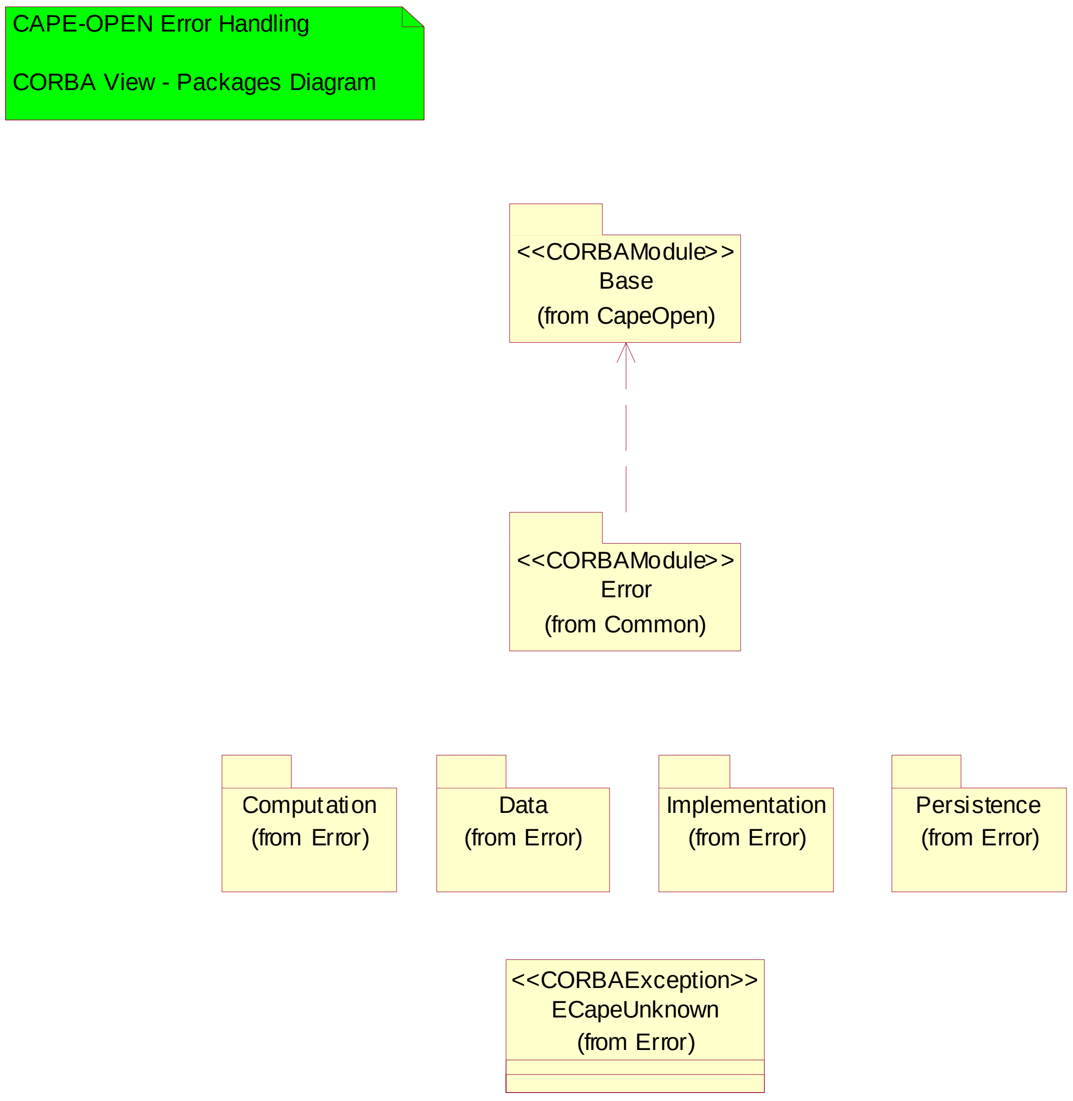

Figure 7 The package dependencies diagram

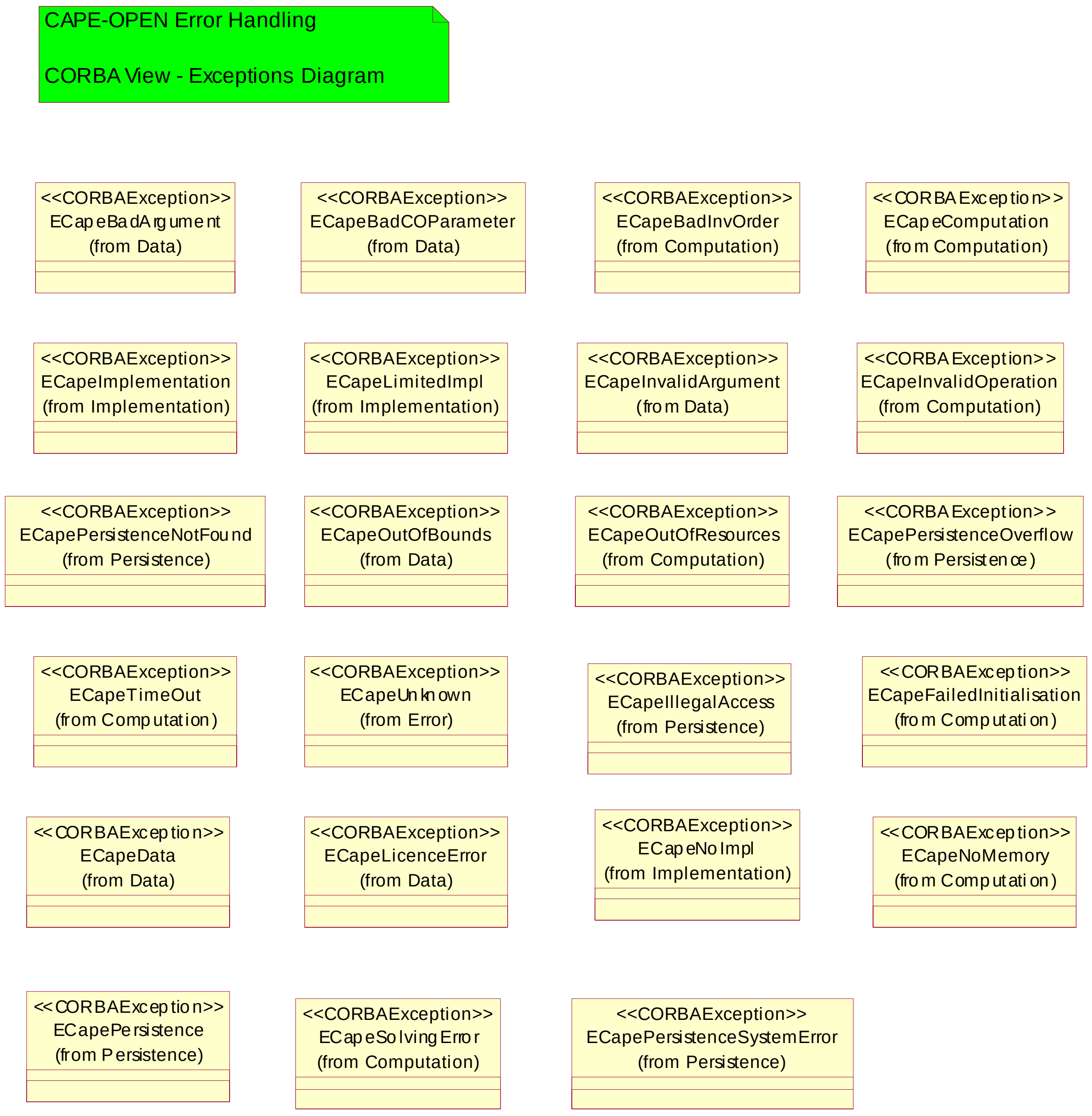
CAPE-OPEN Error Handling
CORBA View - Exceptions Diagram
<<CORBAException>> ECapeBadArgument (from Data)
<<CORBAException>> ECapeBadCOParameter (from Data)
<<CORBAException>> ECapeBadInvOrder (from Computation)
<<CORBAException>> ECapeComputation (from Computation)
<<CORBAException>> ECapeImplementation (from Implementation)
<<CORBAException>> ECapeLimitedImpl (from Implementation)
<<CORBAException>> ECapeInvalidArgument (from Data)
<<CORBAException>> ECapeInvalidOperation (from Computation)
<<CORBAException>> ECapePersistenceNotFound (from Persistence)
<<CORBAException>> ECapeOutOfBounds (from Data)
<<CORBAException>> ECapeOutOfResources (from Computation)
<<CORBAException>> ECapePersistenceOverflow (from Persistence)
<<CORBAException>> ECapeTimeOut (from Computation)
<<CORBAException>> ECapeUnknown (from Error)
<<CORBAException>> ECapeIllegalAccess (from Persistence)
<<CORBAException>> ECapeFailedInitialisation (from Computation)
<<CORBAException>> ECapeData (from Data)
<<CORBAException>> ECapeLicenceError (from Data)
<<CORBAException>> ECapeNoImpl (from Implementation)
<<CORBAException>> ECapeNoMemory (from Computation)
<<CORBAException>> ECapePersistence (from Persistence)
<<CORBAException>> ECapeSolvingError (from Computation)
<<CORBAException>> ECapePersistenceSystemError (from Persistence)

Figure 8 The exception diagram

5.3 Guidelines for CO interface designers

5.3.1 Common errors

This section describes how the CO interface designer deals with the errors which are common errors and defined by this document.

The CO interface designer is in charge of specifying for each CO operations the error(s) that the operation can raise. The interface specification document identifies the exceptional conditions at different levels:

- The analysis view: the use-cases with the item "Exceptions: list of exceptions possibly during execution of the use case". At this level, there is only a textual description of the exceptional condition. It is not indicated to show explicitly the type of the error class ECapeXxxx described by the section Analysis and Design.

- The design view: the interface diagram with the dependency relationship "send" between the operation and the error classes ECapeXxxx described by the section Analysis and Design. This way is difficult to draw. So this representation is not mandatory.

- The design view: the interface description with the item "Errors/Exceptions". As this description is independent on the system, COM or CORBA, this list encloses the probable error classes ECapeXxxx described by the section Analysis and Design. This list can not have any abstract error classes. This list must include the ECapeUnknown error. More over the ECapeInvalidArgument error must be included if there is at least one argument.

- The specification view: the CORBA IDL file. The operations define the corresponding CORBA exceptions using the IDL keyword “exception”. These potential exceptions come from the section Resulting design. They belong to the scope CapeOpen::Common::Error.

Example:

```
void CalcEquilibrium(in Cose::ICapeThermoMaterialObject matObj, in Base::CapeString
flashType, in Base::CapeStringSequence props) raises (Common::Error::ECapeUnknown,
Common::Error::ECapeInvalidArgument, Common::Error::ECapeSolvingError,
Common::Error::ECapeOutOfBounds, Common::Error::ECapeLicenceError);
```

- The specification view: the COM IDL file. The operations define the corresponding COM exceptions as a comment before each operation definition. These potential exceptions come from the section Resulting design.

Example:

```
/*
Exceptions: ECapeUnknown, ECapeInvalidArgument, ECapeSolvingError, ECapeOutOfBounds,
ECapeLicenceError
*/
HRESULT CalcEquilibrium([in] ICapeThermoMaterialObject* matObj, [in] CapeString
flashType,[in]CapeVariant props);
```

As an example on how to respect the Error handling strategy, you can refer to the Open Interface Specification: Parameter Common Interface and Open Interface Specification: Identification Common Interface documents.

5.3.2 Specific Errors

This section describes how the CO interface designer deals with the errors which are not common errors. They are not defined by this document. Indeed the CO interface designer can always define CO error(s) specific to her/his interface. This specific error is designed from the conceptual view, that leads to an exception for CORBA and an error interface for COM.

The CO interface designer can not define abstract error class. The specific error belongs to the scope defined by the designed interface.

Additionally to the common errors (see the previous section), the CO interface designer is in charge of specifying for each CO operations the specific error(s) if any that the operation can raise. The interface specification document identifies the exceptional conditions at different levels:

- The analysis view: the use-cases with the item "Exceptions: list of exceptions possibly during execution of the use case". At this level, there is only a textual description of the exceptional condition.

- The design view: the place of specific error ECapeYyyyXxxx (Yyyy is the name of the interface scope for instance Unit, Ppdb, Smst) within the arborescence defined in the section Analysis and Design as well as its description.

- The design view: the interface diagram with the dependency relationship "send" between the operation and the specific error classes ECapeYyyyXxxx. This way is difficult to draw. So this representation is not mandatory.

- The design view: the interface description with the item "Errors/Exceptions". As this description is independent on the system, COM or CORBA, this list encloses the probable specific error classes ECapeYyyyXxxx. This list can not have any abstract error classes.

- The specification view: the CORBA IDL file. The operations defines the corresponding CORBA specific exceptions. They respect the mapping detailed in the section Technical information. They belong to the scope of the interface (for instance CapeOpen::Ppdb).

Example:

```
Module Ppdb{
…
// This error derives from Common::Error::ECapePersistence
exception ECapePpdbMyError{
             Base::CapeLong code;
             Base::CapeString description;
             CapeCompletionStatus status;
             Base::CapeString scope;
             Base::CapeString interfaceName;
             Base::CapeString operation;
      Base::CapeURL moreInfo;
      Base::CapeString myField; //specific field
};
…
interface ICapePpdbOperation{
      void MyOperation() raises (Common::Error::ECapeUnknown, ECapePpdbMyError);
      void MyOperation2(in Base::CapeString reason) raises (Common::Error::ECapeUnknown,
Common::Error::ECapeInvalidArgument, ECapePpdbMyError);
};
…
};
```

- The specification view: the COM IDL file. The operations define the corresponding COM exceptions as a comment before each operation definition. They respect the mapping detailed in the section Technical information. As already mentioned in section Resulting design, the CO interface designer must request a new value for her/his specific error to the M&T group.

Example:

```
// This error is derived from ECapePersistence
/*
Value: OLE_E_FIRST + (an error number approved by the M&T group)
*/
interface ECapePpdbMyError: ECapePersistence{
      CapeString myField; //specific field
}
…
interface ICapePpdbInterface:IDispatch{
/*
Exceptions: ECapeUnknown, ECapePpdbMyError
*/
HRESULT Operation();
}
```

6. Prototypes implementation

6.1 COM Implementation

6.1.1 Client

This is the prototype source code for clients which requests a CO operation from a COM CAPE-OPEN component. When calling the operation, the client must check if the component raised a CAPE-OPEN error.

C++ CLIENT PROTOTYPE

```
#include "testErrClient.h"

#import "CAPE-OPENv0-93.tlb" no_namespace named_guids
#include "comutil.h"
void testErr();
int main(int argc, char* argv[])
{
      testErr();
}

void testErr()
{
CLSID         myCLSID;
_bstr_t       name, errDescription, errSource;
long          pos, errCode;
BSTR          bstr;
HRESULT       hr                  = S_FALSE;
ICapeParameter* pIdent            = NULL;

CoInitialize(0);
CLSIDFromProgID(L"TestErrServer.CTestErrServer", &myCLSID);
hr = CoCreateInstance(myCLSID, 0, CLSCTX_ALL, IID_ICapeParameter, (void**)&pIdent);

SetErrorInfo(0L, NULL);
//wrongly passing a pointer reference instead of a valid parameter value

variant_t v;
v = pIdent;
hr = pIdent->put_value(v);
if (SUCCEEDED(hr)) {
;//success
} else {
switch (hr) {
case ECapeBadArgumentHR:
//ECapeBadArgumentHR inherits from ECapeUser (and this from ECapeRoot)
      ECapeBadArgument *pBadArg;
      hr= pIdent->QueryInterface(IID_ECapeBadArgument,
(void**)&pBadArg);
  pBadArg->get_position(&pos);
  //ECapeUser members:
ECapeUser* pUser;
hr= pIdent->QueryInterface(IID_ECapeUser,(void**)&pUser);
hr = pUser->get_description(&bstr);
hr = pUser->get_code(&errCode);

errDescription = bstr;
               //ECapeRoot members
ECapeRoot* pRoot;
hr= pIdent->QueryInterface(IID_ECapeRoot,(void**)& pRoot);
hr = pRoot ->get_name(&bstr);

break;
```

```
             /* Add here support for the rest of eCapeErrorInterfaceHR_tag values
corresponding to the list of exceptions that method put ComponentName may raise.
             */
default:
//it's not a CO exception. So, we treat as standard COM
IErrorInfo* pErrinfo;
hr=GetErrorInfo(0L, &pErrinfo);
hr=pErrinfo->GetDescription(&bstr);
errDescription = bstr;
pErrinfo->GetSource(&bstr);
errSource= bstr;
}
}
::SysFreeString(bstr);
}
```

VB CLIENT PROTOTYPE

```
Private Sub testCOError_Click()
 Dim testServer As ICapeParameter
    Dim capeBadArg As ECapeBadArgument
    Dim vs As Long

    Set testServer = CreateObject("TestErrServer.CTestErrServer")
On Error GoTo testErrors
    'wrongly passing a pointer reference instead of a valid parameter value
 testServer.Value = testServer


    Exit Sub
testErrors:
    If Err.Number = capeopen.ECapeBadArgumentHR Then
        Set capeBadArg = testServer
        MsgBox ("position of wrong argument:" & capeBadArg.position)
        ‘ checking the members of ECapeBadArgument’s parent exceptions
        Dim capeRoot As ECapeRoot
        Set capeRoot = testServer
        MsgBox ("Name of raised exception" + capeRoot.Name)
    else if ...
‘      Add here support for the rest of eCapeErrorInterfaceHR_tag values ‘corresponding
to the list of exceptions that method put ComponentName may ‘raise.
  else
       ‘Check non CAPE-OPEN errors through VB’s Err keyword
  end if

    End Sub
```

6.1.2 Server

This is the prototype source code for COM CAPE-OPEN component. When calling the operation, the client must check if the component raised a CO error.

C++ SERVER

```
class CTestErrServerCpp :
       public IDispatchImpl<ITestErrServerCpp, &IID_ITestErrServerCpp,
&LIBID_TESTERRSERVERLib>,
       public ISupportErrorInfo,
       public CComObjectRoot,
       public CComCoClass<CTestErrServerCpp,&CLSID_TestErrServerCpp>,
       public IDispatchImpl<ICapeParameter, &IID_ICapeParameter, &LIBID_CAPEOPEN>,
       public IDispatchImpl<ECapeRoot, &IID_ICapeRoot, &LIBID_CAPEOPEN>,
public IDispatchImpl<ECapeUser, &IID_ECapeUser, &LIBID_CAPEOPEN>,
public IDispatchImpl<ECapeBadArgument, &IID_ECapeBadArgument, &LIBID_CAPEOPEN>

{
public:
       CTestErrServerCpp() {}
BEGIN_COM_MAP(CTestErrServerCpp)
```

```
//DEL  COM_INTERFACE_ENTRY(IDispatch)
       COM_INTERFACE_ENTRY(ITestErrServerCpp)
       COM_INTERFACE_ENTRY(ISupportErrorInfo)
       COM_INTERFACE_ENTRY2(IDispatch, ITestErrServerCpp)
       COM_INTERFACE_ENTRY(ICapeParameter)
       COM_INTERFACE_ENTRY(ECapeRoot)
COM_INTERFACE_ENTRY(ECapeUser)
COM_INTERFACE_ENTRY(ECapeBadArgument)
END_COM_MAP()
//DECLARE_NOT_AGGREGATABLE(CTestErrServerCpp)
// Remove the comment from the line above if you don't want your object to
// support aggregation.

DECLARE_REGISTRY_RESOURCEID(IDR_TestErrServerCpp)

// ICapeParameter
STDMETHOD(put_value)(VARIANT value)
{
    if (V_VT(&value) == VT_DISPATCH) {
        m_ m_ECapeRootName = "Bad Argument type"
        m_ECapeBadArgumentPos = 1;
        return ECapeBadArgumentHR;
    }
    //normal code....
              return E_NOTIMPL;
} //.. rest of ICapeParameter methods

//ECapeRoot
CString m_ECapeRootName;

STDMETHOD(get_Name)(BSTR * name)
{
       if (name == NULL)
              return E_POINTER;
       *name = m_ECapeRootName.AllocSysString();

       return S_OK;
}
//ECapeUser
       //implement here ECapeUser’s methods

// ECapeBadArgument
long m_ECapeBadArgumentPos;
STDMETHOD(get_position)(LONG * position)
{
       if (position == NULL)
              return E_POINTER;
       *position = m_ECapeBadArgumentPos;
       return S_OK;
}
```

VB SERVER

```
Implements ICapeParameter

Implements ECapeBadArgument
Implements ECapeUser
Implements ECapeRoot
'not actually required because they don’t have any method
Implements ECapeData



Private Property Get ECapeBadArgument_position() As Long
‘Value property only has 1 argument
ECapeBadArgument_position = 1
End Property

Private Property Get ECapeUser_description() As String
```

```
End Property

Private Property Get ECapeRoot_name() As String
    ECapeRoot_name = "Testing CO Error Service"
End Property

‘ICapeParameter
Private Property Let ICapeParameter_Value(ByVal RHS As Variant)
    If VarType(RHS) = vbObject Then
Call Err.Raise(capeopen.ECapeBadArgumentHR)
    End If
End Property
‘... rest of ICapeParameter methods
```

6.2 CORBA Implementation

The following code is Java code. Of course, the way to handle CO errors respects IDL to Java Language Mapping Specification, OMG, June 1999.

6.2.1 Client

This part of simplified code shows how a client side application calls CO operations from a CAPE-OPEN component and checks if a CO error is raised. This example concerns a constant temperature jacket reactor program that solves its mathematical description using a CO solver component.

```
private void ctjrDAE(ORB orb, String url) {
      try{

      // Creation of the model by means of ModelManager

      // Creation of the DAE ESO by means of ESOManager

      // Build the DAE ESO

      // Associate the DAE ESO to the Continuous model

      // Create the Event

      // Add an external event info containing the event to the model

      // Locate and bind to the remote NSP application (a CO compliant solver component)

      // Create a DAE Solver and solve the CTJR problem
      ICapeNumericDAESolver _solver = smanager.CreateSolver(CapeSolverType.DAE, _model);

      _solver.SetComponentName("NSP DAE Solver from LSODIR legacy code");

      double tols[]=new double[size];
      for (int i=0; i<size; i++) tols[i]=1E-6;
      _solver.SetRelTolerance(tols);
      _solver.SetAbsTolerance(tols);

      _solver.Solve();
      double solution[] = _solver.GetSolution();

      } catch (org.omg.CORBA.UserException e){
         System.out.println(" !!!! CO Error occured : ");
         System.out.println(e.toString());

      } catch (Exception e){ // other exceptions such as CORBA system exceptions
         e.printStackTrace();

      } finally {
         // Destroying the remote solver
         try{
            if (_solver!=null) _solver.Destroy();
         } catch (org.omg.CORBA.UserException e){
           System.out.println(" !!!! CO Error occured : ");
           System.out.println(e.toString());
         }
      }

   }
```

6.2.2 Server

This part of simplified code shows how an implementation of CO operation in a CAPE-OPEN component can deal with CO errors. This example concerns the ICapeNumericSolverManager::CreateSolver operation from a CO solver component.

```
public ICapeNumericSolver CreateSolver(CapeOpen.Numr.Solver.CapeSolverType type,
CapeOpen.Numr.Model.ICapeNumericModel theModel) throws
ECapeUnknown,ECapeInvalidArgument,ECapeOutOfBounds,ECapeOutOfResources {

   if (_solverPOA==null) throw new ECapeUnknown(1000, "POA is null: the solver object can
not be created", "CapeOpen::Numr::Solver", "ICapeNumericSolverManager", "CreateSolver",
"None");
   if (theModel==null) throw new ECapeInvalidArgument(1000, "The model is null: it is
useless to create a solver object since the model can not be set later",
"CapeOpen::Numr::Solver", "ICapeNumericSolverManager", "CreateSolver", "None", (short)2);

   ICapeNumericSolver solver=null;

   switch (type.value()){
     case CapeSolverType._LA:
     ICapeNumericLASolverPOATie tieLA = new ICapeNumericLASolverPOATie(new
LASolver(theModel));
     if (tieLA==null) throw new ECapeUnknown(1000, "Creation of LASolver failed",
"CapeOpen::Numr::Solver", "ICapeNumericSolverManager", "CreateSolver", "None");
     try{
        _solverPOA.activate_object(tieLA);
        solver =
ICapeNumericLASolverHelper.narrow(_solverPOA.servant_to_reference(tieLA));
                 ((LASolver)tieLA._delegate()).completeAOMdata(_solverPOA, tieLA);
     } catch (Exception e){
       e.printStackTrace();
       throw new ECapeUnknown(1000, "An error occurres when the LA Solver is created",
"CapeOpen::Numr::Solver", "ICapeNumericSolverManager", "CreateSolver", "None");
     }

       break;

       case CapeSolverType._NLA:
       ICapeNumericNLASolverPOATie tieNLA = new ICapeNumericNLASolverPOATie(new
NLASolver(theModel));
       if (tieNLA==null) throw new ECapeUnknown(1000, "Creation of NLASolver failed",
"CapeOpen::Numr::Solver", "ICapeNumericSolverManager", "CreateSolver", "None");
       try{
          _solverPOA.activate_object(tieNLA);
          solver =
ICapeNumericNLASolverHelper.narrow(_solverPOA.servant_to_reference(tieNLA));
                 ((NLASolver)tieNLA._delegate()).completeAOMdata(_solverPOA, tieNLA);
       } catch (Exception e){
          e.printStackTrace();
          throw new ECapeUnknown(1000, "An error occurres when the NLA Solver is
created", "CapeOpen::Numr::Solver", "ICapeNumericSolverManager", "CreateSolver", "None");
       }
       break;

       case CapeSolverType._DAE:
          ICapeNumericDAESolverPOATie tieDAE = new ICapeNumericDAESolverPOATie(new
DAESolver(theModel));
          if (tieDAE==null) throw new ECapeUnknown(1000, "Creation of DAESolver failed",
"CapeOpen::Numr::Solver", "ICapeNumericSolverManager", "CreateSolver", "None");
          try{
             _solverPOA.activate_object(tieDAE);
         solver = ICapeNumericDAESolverHelper.narrow(_solverPOA.servant_to_reference(tieDAE));

                  ((DAESolver)tieDAE._delegate()).completeAOMdata(_solverPOA, tieDAE);
          } catch (Exception e){
             e.printStackTrace();
             throw new ECapeUnknown(1000, "An error occurres when the DAE Solver is
created", "CapeOpen::Numr::Solver", "ICapeNumericSolverManager", "CreateSolver", "None");
          }
        break;

        default: throw new ECapeInvalidArgument(1000, "The type of the solver is not
correct", "CapeOpen::Numr::Solver", "ICapeNumericSolverManager", "CreateSolver", "None",
(short)1);
      }

      return solver;

   }
```

7. Specific glossary terms

8. Bibliography

The bibliography is organised in three sections as shown below.

8.1 Process simulation references

8.2 Computing references

(i) The UML User Guide, G. Booch - J. Rumbaugh - I. Jacobson , Addison-Wesley, 1998

(ii) The Common Object Request Broker: Architecture and Specification 2.3.1, OMG, October 1999

(iii) C++ Language Mapping Specification, OMG, June 1999

(iv) IDL to Java Language Mapping Specification, OMG, June 1999

(v) Integrating CORBA and COM applications, M. Rosen - D. Curtis, Wiley

8.3 General references

8.3.1 CAPE-OPEN project references

(i) Open Interface Specification : Unit Operation

(ii) Open Interface Specification : Thermodynamic and Physical Properties

(iii) Open Interface Specification : Numerical Solvers

(iv) Open Interface Specification : Sequential Modular Specific Tools

8.3.2 GLOBAL CAPE-OPEN project references

(i) Methods and Tools Integrated Guidelines

(ii) Open Specification Interface: Identification Common Interface

(iii) Open Specification Interface: Parameter Common Interface

(iv) CAPE-OPEN Type Library: CAPE-OPENv0-9.tlb

(v) Proposal on a "full" consistent CO CORBA Specification

(vi) CAPE-OPEN IDL Library: CAPE-OPENv0-9-1.idl

(vii) CAPE-OPEN IDL Library: CAPE-OPENv0-9-2.idl

9. Appendices