Exception Handling in CORBA Environments

The Late Introduction of Distributed Exception Handling in Java™ CORBA–Based COTS Application Domains

Component–based software development introduces new sources of risk because (i) independently developed components cannot be fully trusted to conform to their published specifications and (ii) software failures are caused by systemic patterns of interaction that cannot be localized to any individual component. The need for a separate exception handling infrastructure to address these issues becomes the responsibility of the exception handling subsystem. COTS components focus on executing their own *normal* problem solving behavior, while their exception handling service focuses on detecting and resolving exceptions within the local COTS domain [Dellarocas 98] The exception handling architecture of the integrated system is realized by adding exception handling logic to each application component using a *middleware* approach.

Catching errors at compile time is the ideal. The rest of the problems must be caught at run–time through some formality that allows the creator of the error to pass appropriate information to the recipient to know how to handle the error. Adding this architecture to an existing system can be a daunting task. A layered approach to detecting and resolving exceptions provides an *incremental* deployment strategy that allows exception handling to be added to the application components in small increments, without disrupting the existing code base.

This paper describes the strategy for constructing of a Java / CORBA exception handling system late in the development cycle that can be extended across the CORBA object boundaries.

Glen B. Alleman galleman@niwotridge.com
8 March 2000, Revised 12 October 2000

Copyright ©, 2000, 2001

Table of Contents

TABLE OF CONTENTS **2**

TABLE OF FIGURES **3**

INTRODUCTION **1**

THE PROBLEM 1
SYSTEMATIC FAILURES IN THE LARGE 1
Exception Handling "After the Fact" *1*
The Context of Exception Handling *1*
Formal Exception Handling Rules *1*
Lack of Formal Exception Specifications *1*
SCOPE OF AN EXCEPTION HANDLER 2
Current Exception Handling Architecture *2*
Deployment of the Exception Handler *2*
THE PURPOSE OF EXCEPTION HANDLING 2
Myths of Exception Handling *2*
Exception Handling Rasion Dêtat *3*
THE EXCEPTION HANDLING PROBLEM 3
The Problem Domain *3*
Subsystems of the Problem *3*
Extending the System Architecture *4*
Dynamic versus Static Exception Handling *4*
THE SOLUTION 5
Some Patterns in High Availability Systems *5*
Degradation of Exception Handling Structure *6*
GENERAL GUIDELINES 6
SYSTEM BEHAVIOR 7
Exception Categories *7*
Exception Handling Approaches *8*
Defining the Exception Classes *8*
Exception Handling in Java *9*
Exception Interfaces *9*
USERS OF THE ERROR HANDLING SYSTEM 9
EXCEPTION HANDLING PROCESS 10
SOME GENERAL COMPONENTS OF THE ERROR HANDLING SYSTEM 11
PRINCIPLES OF THE ERROR HANDLING SYSTEM 10
ADDRESSING THE ERROR HANDLING REQUIREMENTS 10
ERROR HANDLING SYSTEM DESIGN PROCESS 10
Exception Handling in CORBA Applications *11*
EXCEPTION HANDLING DESIGN NOTES 11
Defining Exceptions *11*
Handling Exceptions *11*
Handle or Declare an Exception? *12*

A SIMPLE ERROR HANDLING ARCHITECTURE **12**

COMPONENTS OF THE ERROR HANDLING SUBSYSTEM 13
A SIMPLE SEQUENCE FOR ERROR HANDLING 13

THE BIG PICTURE **14**

EXCEPTION HANDLING WITHIN CORBA **17**

REPORTING EXCEPTIONS ACROSS CORBA LINKS 17

REFERENCES **17**

Table of Figures

Figure 1 – Exception Detection Conditions 3
Figure 2 – Error Handler Subsystems 5
Figure 3 – Behaviors of a less than perfect system 7
Figure 4 – Three Exception Models 9
Figure 5 – The Various Participants in the Error Handling System 10
Figure 6 – The Causal Relationships in an error handling system 11
Figure 7 – The Structure of the Error Handling Subsystem 12
Figure 8 – Simple Exception Handling Sequence 13
Figure 9 – Master Class Diagram for the Error Handling Subsystem 16

Introduction

The Golden Rule of Programming: Errors Happen

The Problem

Complex COTS–based products are assembled from components provided by other systems. These components encapsulate parts of the other system's data and behavior. and are accessed through public interfaces. Exceptions and exception handling are part of these external interfaces and therefore must be included in behavior of the integrated COTS system in some manner [Romanovsky 99].

Exception handling is a structuring mechanism that allows the separation of normal behavior from abnormal behavior in the software system [Christian 94], [Goodenough 75].

Exception handling separates the code for normal execution from the code for exception handling. It separates normal flow of control from the exception flow of control. In practice, components have sophisticated means of handling exceptions that are propagated across the external interface boundary of the component. Any component using other components must therefore be aware of these exceptions and be capable of handling them when they appear.

Systematic Failures in the Large

The CORBA exception handling domain is a *programming in the large* problem. Business processes can be integrated by assembling heterogeneous CORBA based components in a distributed environment. Exception handling is provided by the operating systems and programming languages, but the assembled system provides little support for exception handling *in the large*.

Exception Handling "After the Fact"

In many instances the exception handling architecture of a system is provided *after the fact*, that is, exception handling at the macro level is introduced after the system components have been selected and integration has begun. In this *late binding* method there are several failure modes of the integrated system that generate exceptions:

- Program failures – that lead to unsuccessful attempts to use a specific computing resource.
- Design and System failures – that lead to unsuccessful invocations of computing resources.
- Communication failures – that prevent to invocation of a specific computing resource.

The Context of Exception Handling

An exception is an unusual event, erroneous or not, that is detectable either by hardware or software and that may require special processing [Sebesta 96].

How can a program fail? Since there are a nearly uncountable number of ways a program can fail, a better question is how can a program be constructed to succeed? The simple answer is that every step of the program's construction must be *correct*. Since this is a nearly impossible task in today's rapid development environment, the program must protect itself from errors, no matter what their source. In CORBA applications, the *program* is made up of distributed objects.

There are several *core* problems in the CORBA based systems that are not currently addressed in the OMG standards [Zinky 97]:

- Most programs are written ignoring the *wide are* conditions of the operational environment.
- When programs attempt to handle these conditions, they encounter difficulty, since these external conditions are much different from the conditions the local objects deal with.
- IDL hides information about the tradeoffs any implementation details an object must make.
- There is currently no way to systematically reuse components that deals with these special conditions, so code sharing becomes difficult.

Formal Exception Handling Rules

To define the rules for handling exceptions and the propagation of exceptions, exception handling techniques must result in the *structuring of exception objects*. A set of exceptions and exception handlers are associated with an *exception context*. If the exception cannot be handled within the specified context, or if there is no handler for the exception, the exception is propagated to the continuing context where the corresponding exception handler can be called.

Each exception handling context is associated with a component or object. In the UML notation, *uses* means that the component refers to the interface or another component. If a dynamic system is considered, where structuring is based on nested method calls, the exception object becomes a full participant in the UML description of the system.

Lack of Formal Exception Specifications

In the absence of formal exception handling requirements, an application must be capable of maintaining an object's consistent state in two ways:

- Ensuring there is a consistent base state for the object when it is constructed. This is done by creating an initialization method, that places the object in its base state after construction, or whenever it needs to be reinitialized.
- The object must maintain a consistent state for each method. This is difficult, since in most cases the states of the object cannot be formally specified.

Given these two conditions, it is now clear why all software systems have errors –

> No one has the time, the patience, or the skills to enumerate all the possible states of an object as well as their accompanying error conditions.

The first approach is usually to inserting code at every step in the method to deal with error conditions. Over time, this code becomes polluted as new or unaccounted for error conditions are discovered.

A new paradigm for handling errors is needed that allows the programmer to develop code for the *normal* case, then hand off any exceptions to normal cases to a separate set of code–fragments. This *exception handling* code is maintained separately from the method's mainline functionality. The disruption to the method's control of flow is minimized, the error handling code is *decoupled* from the method, and the *cohesion* between exception handling and functional code is minimized [Parnas 72]

Scope of an Exception Handler

The Exception Handler must be applied to all domains of the system, including C++ and Java™ code. In the context of this paper, the communication mechanism for the exception handler will make use of the serializable IO of Java™ and CORBA.

Current Exception Handling Architecture

The COTS nature of many applications creates an exception handling architecture that is focused on the isolated domains of the COTS components, rather than the domain of the *federated* system. This approach is appropriate for short development schedules, time to market pressures, and the limited resources all focused on the *first product release*. As the product matures, a more sophisticated exception handling architecture must evolve in order to sustain the benefits of the COTS based product strategy.

Deployment of the Exception Handler

This approach to exception handling would be considered *ad hoc* at best. This is not due to the lack of architectural consistency, poor object decomposition, or even poor design. It is the direct result of schedule and resource limitations. In theory, exception handling is part of the software development activity. In practice, exception handling is an architectural level activity first, followed by design and implementation. The design and implementation activities produce products on short schedules. The architectural aspects of the system are *firm*, in that the object interaction architecture is usually in place long before the actual objects exists. In many cases, the details of the exception handling processes for each object are defined and implemented after the fact. There was no way to avoid this process given the circumstances of the rapid development schedule.

> In theory, there is no difference between theory and practice. But in practice, there is.

It is unreasonable to assume that a COTS–based exception handling architecture can be applied in whole to a system. A successful approach is to incrementally introduce the Exception Handler(s) in core components at specific releases of an integrated COTS application.

The Purpose of Exception Handling

It may seem obvious what the purpose of exception handling is, but this is not so. One assumption is that exception handling is the means of catching and reporting something that has gone wrong in the program. However, this is too broad a definition, since some error conditions are normal, while others are exceptional.

Exception handling addresses only those conditions that are not handled by the normal flow of the program, e.g. *exceptions*. This may appear as a tautology, but this fact is often overlooked in the architecture of integrated system.

Myths of Exception Handling

There are some popular myths about exception handling, created through lack of understanding or misunderstanding of the importance of exception handling in a COTS–based environment:

- Exception handling reduces the amount of code to be written.

  Exception handlers do not reduce the amount of code, they simply collect the code in one place and give the programmer greater flexibility about where to place the code.
- Expectation handling is embedded in the code as part of the normal processing.

  By separating the exception handling code from the normal application code, all the attributes of object–oriented programming can be applied to the exception code. The main logic is not cluttered with tests for every possible thing that can go wrong.
- Return values are a viable way of handling exceptions.

Exceptions cannot be ignored, return values can. When an exception is raised, the flow of control is changed whether the programmer has prepared for it or not. If the exception is anticipated, then it will be handled at an appropriate place. If the exception is unanticipated, then the program will stop running before any other errors occur.

Exception Handling *Rasion Dêtat*

> Exception handling exists to provide the means of dealing with unusual conditions that can arise during the execution of a program in a consistent way [Litwak 99].

The Exception Handling Problem

Software systems must provide some form of feedback to the user when an error or exception occurs. This feedback can be in the form of an error message presented to the user, an error message logged on the system console, or an entry in a log file that will be processed later.

In all cases, this feedback is not related to the user interface interaction. These exception messages occur when something has gone wrong with the normal operation of the system. Improper user input or invalid command usage is part of the normal system behavior and is not considered an exception.

The Problem Domain

A large part of any software design is the identification and handling of error and exception conditions [Knudsen 00]. Several of the unique characteristics of distributed CORBA software are its reliability, the human factors of the applications, and the user interaction with these applications. Previous generation mainframe systems were designed to be in continuous operation with the requirement that they be out of service no more than a few hours in a year. This requirement in many cases limits the design choices that can be made in the modern distributed processing environment.

Subsystems of the Problem

In order to *partition* the problem into manageable components, some form of subsystem decomposition must be made for the exception handling architecture. Starting with the Exception Handler two subsystems can be identified, as shown in Figure 2:

- Exception Handling – which provides the detection, isolation, and reporting of errors that occur during the operation of the application. Error handling consists of three layers of exception processing.
  - Exception Detection – before an error can be handled it has to be detected. The run–time code must be enriched before proper error detection can be achieved. Failures can only be detected in relation to specified behavior of the component. The specification itself is assumed correct.

The method of exception detection is to instrument nearly every method to verify the state and behavior of the system at run–time. Writing precise pre–conditions, post–conditions, and invariants for each method is the *correct* way to do this. Typically, these pre and post condition will include the items show in Figure 1.

This list can be extended by individual checks for special assertions, for example loop invariants. The pre– and post–conditions are constraints of the class' internal state and the state of classes form the environment, derived from the specification.

In the initial implementation of most integrated applications, the use of assertions is *light* as best. Adding assertions to the code is a tedious task, since both the specification for the assertion and the code to handle the violation of the assertion must be written.

One approach for the future is an *assertion–checking object* that can be tuned on or off depending on the need (development or production).

What ?	Where?
Invalid Parameters	On entry to the method
Violation of the precondition	On entry to the method
Unexpected results to failures of methods called by a client method.	Immediately after return of the method, the appropriate exception handler is invoked.
Violation of a method's invariants.	There are two kinds of invariants: (1) some invariant that is related to the whole classes that are checked at the end of every method of that class, (2) some invariant that is shared between classes (aliasing). These invariants must be checked on entry as well as exit form the method.

Figure 1 – Exception Detection Conditions

  - Isolation – isolating exceptions must take place within the context of the methods that created the exception or the immediate client that invoked the method. Once the exception has been forwarded (without additional information about the context) the exception semantics become overloaded.

The first approach to isolating exceptions is to *fix* the error at the source of the problem. This usually involves adding code

at the location of the exception. In many instances, this is not possible since the code may only be supplied in executable form.

- Reporting – the choice to report an exception to a human is dependent not only on the exception but also on the ability to resolve to exception. In general, exceptions are not reported to humans, but are used by programs to resolve the exception or terminate the execution of the program.

Another approach is that exceptions are not reported at all to humans in the sense of a *user*. The system does interact with its users and reports *errors* in the users input to the system, but no exceptions are signaled to users, just exception handling methods.

This follows the design pattern of *checks* and reserves exceptions for errant methods not invalid user input [Cunningham 95].

- User Feedback – which provides the visual means to reporting errors to the User or System Administrator. Following the design rules in the above paragraph., user feedback is provided only when a human can remedy the situation.
- Input Verification – which provides a standard means of verifying input data presented to the system before it causes and error. This is another area where exceptions are not normally used, but errors are communicated to the user through a dialog process.

Extending the System Architecture

In order to provide for error handling and reporting, a system architecture must be extended in the following manner:

- Error Detection – using standard software idioms and conventions for detecting, throwing and catching, and forwarding errors.
- Error Handling – through retry, organized panic, and alarming processes.
- Propagation of Errors – through an Error Object that is created for the specific instance of the error.
- Administration and collection of error information – through the error logging and display processes built into the error handling system.
- Administration of the error messages – through a centralized Error Object management system.

Dynamic versus Static Exception Handling

There are two basic paradigms for handling exceptions in modern programming languages [Knudsen 00]:

- Static exception handling – in which exceptions are handled by objects and code fragments designed into the application. In the static exception model, the exception handling code is designed to match the enumerated exceptions, either through an exception list interface or an enumeration of exceptions within a code section.

  There are several issues associated with Static exception handling, including:

  - Failure to enumerate all the exceptions – through either oversight or intention.
  - Failure to control the creation of all objects in the application – the assumption that all the code in the system is under the control of the development team may be naï ve. The result is a set of objects that are not created by the running application, but may have be created by other applications and made available to the current application.
- Dynamic exception handling – in which exception objects are created, thrown, and handled using try blocks.
  - Exception Objects – the purpose object is to act as a messenger between the point where the exception occurrence and the location where the exception is handled.
  - Throwing Exceptions – when an exception has been identified, the dynamic exception handling system throws an exception object to the exception handler. The exception object travels back along the dynamic call chain until an exception handler designated for the particular exception object is located. At this point, the exception handler gains access to the exception object. The proper exception handling processing is then applied to this object, based on the information found in the object. At this point, the exception handler can decide how to properly continue the execution of the application.
  - Try Blocks – provide the means for specifying the extent of the exception handlers. The try block is capable of handling the exceptions for which a defined handler exists. It is during the exception handling of the try block that an exception catch clause is located.
  - Exception Handlers – is located within a catch clause. During the execution of the catch clause, access to the exception object is provided. Special processing information may be placed in the exception object for processing further up the call chain.

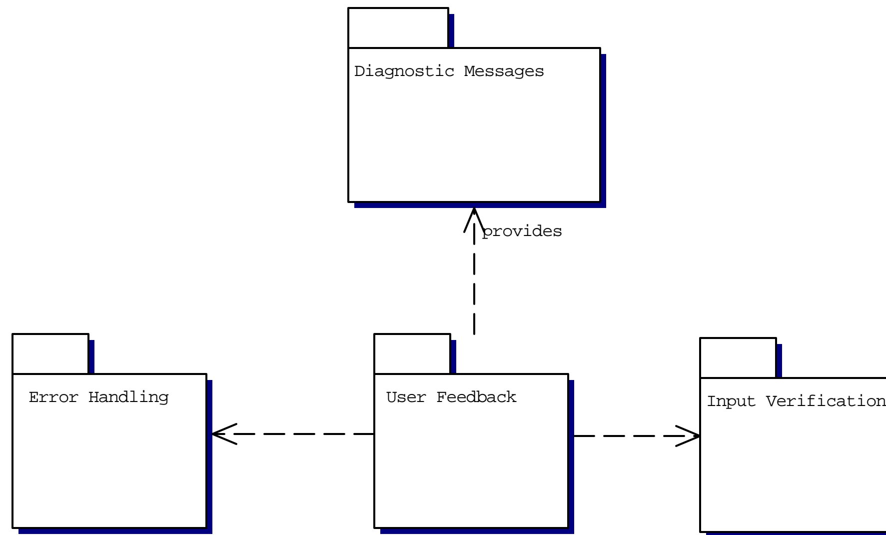

Figure 2 – Error Handler Subsystems

The Solution

The technical issues raised in this document and addressed in the proposed design describe an *embedded* exception handling subsystem for a COTS integrated product. Such an exception handling system maintains the integrity of the system components outside of the exceptions encountered *within* the individual components.

> The basic philosophy of Java™ is that *badly formed code will not run* [Eckel 98].

Some Patterns in High Availability Systems

The following is a brief list of patterns occurring in the design of high availability systems. These patterns can be applied throughout the exception handling system. These patterns may appear obvious or even trite, but they are derived from several important sources including Lucent Technologies patterns for highly available systems [Coplien 95] [Hammer 98].

- Minimize Human Intervention – history shows that people cause of the majority of problems in computer systems. Problems occur because of wrong actions, wrong configurations, or the wrong button was pushed. The solution is to let the machine do everything possible to remain operational, deferring to humans only as a last resort.
- People Know Best – automation must be balanced with human authority and responsibility. In a high availability environment, the system may not be able to recover from all faults. People have a subjective sense of the importance of external faults, and the actions needed to repair them. The solution is to assume that people know best, particularly the system administrators. The system should allow knowledgeable users to override the automated controls.
- Five Minutes of No Escalation Messages – streaming error messages to the system console saturates the human–machine interface and consumes resources for the display of information. In many cases, transient faults occur that will be repaired by the normal operational software. Given time many problems work themselves out. The solution is to display an error message for the change in state of the system, not the continuous reporting of the current state.
- Riding Over Transients – in many cases the detected problem may be a transient. The question is if the problem will work itself out or not. The solution is not to immediately react to detected conditions. Make sure the condition really exists by checking several times. In many cases, the error can be resolved with a minimum of effort if the error is isolated. The density of the error conditions is important here.
- SICO First and Always – the System Integrity Control Program (SICO) is the core component that provides diagnostics and operational control of the system. This component must be trusted in a way that allows it to control the actions of other system components. This allows the system to be reinitialized whenever the stability of the system is in question. The same system integrity components should oversee both the initialization process and the normal application functionality so that initialization can be restarted if it runs into an error.
- Fool Me Once – sometimes the source of a fault is transient or intermittent. After the detection and recovery from the fault, the user expects the notification of the fault

to disappear. If the display of errors continues for more than a reasonable period of time (30 seconds after the repair for example), the user will become concerned that the system may still have a problem.

Degradation of Exception Handling Structure

Let's assume that the integrated system contains the necessary exception handling services for the initial release. Despite the best intentions of the developers, the exception handling system (as well as all other subsystems) degrade over time. The addition of new objects, changes to existing objects, performance improvements, upgrades to COTS components, all contribute to violations of the desired system architecture.

The degradation of the structure of the objects and their methods has been studied for several years [Belady 76]. Although the degradation of the code associated with the normal flow of control is important to the stability of the system, the exception handling code undergoes the same degradation.

The degradation of the exception handling code is different through. There are a number of causes to this that must be addressed in the application to maintain its *high availability* status:

- Unanticipated Exception Sources – the full set of exceptions that can arise in the system cannot be anticipated. The late determination of exceptions makes it difficult to design and implement a well–structured set of exception handlers and propagation policies for the various kinds of exceptions.
- Unanticipated Exceptions – Java™ supports both checked and unchecked exceptions. The use on unchecked exceptions leads to two problems:
  - Pervasive exception types – such as `NullPointerException` can circulate freely in a program, sometimes reaching the entry point of the application and cause the program to crash. The problem of uncaught exceptions is addressed in [Fahndrich 98], [Pessaux 99], [Robillard 99], [Yi 98], [Yi 97]
  - Subsumed exceptions by more general types – the type `Exception` can overload the exception handlers with vague types.
- Handler Overload – an exception handler in Java™ states the types of exceptions it handles. Since Java™ exceptions are related to the type hierarchy system, through subsumption the exception handler may end up handling more than one kind of exception [Robillard 99]. Handler overload is unavoidable if the developer raises an explicit exception of a type that is commonly defined as the super–type of other exceptions, e.g. `RunTimeError`. In this case it will be impossible for clients of the method that raised the exception to handled the exception specifically.
- Propagation Of Exceptions – every time an exception is propagated, it loses context. For example when a method reads from a stream object that it received as an argument, an `IOException` may occur. The method typically cannot recovery from this exception because it did not create the stream object and does not know the name of the source file used to create the stream object. The method then has to propagate the exception. It is unlikely the client of the method can handle the exception either. The lack of sufficient context makes it difficult for clients to design good recovery or useful notification when catching propagated exceptions.
- Exception Overload – exceptions can become semantically overloaded when a client introduces an exception handler for an exception raised by a component that itself is changing or being extended in ways no defined by the original exception semantics. This overloading may degenerate the meaning of a particular exception type for a component.

  The explicit declaration of the exception semantics is needed to avoid the semantic overloading of the exception results. For example,

```
Void f() throws tooBig; tooSmall; divZero { //…
```

  Declares the explicit exceptions that are thrown by the function `f`. These exceptions are members of the exception class defined by Figure 9, which are in turn members of the built–in Java™ exception class.
- Systematic Ignoring Of Exceptions – in some cases the catching of an exception and doing nothing is found in the code. The program ends up in an inconsistent state when this happens. Searching for these `catch` approaches is a high priority for the integration components
- Unspecified Exception Values – a Java™ exception carries a value. One use of the value is to store an explanatory `string` that can be used to describe the exception condition. The value of the string is set when the exception is created. Since this assignment depends on the developer for the correct wording. This is a source of confusion and possible exception handler errors.

```
If (t == null)
     throw new NullPointerException ("t = null");
```

  It is unlikely that this type of exception handling is being done throughout the Insight code. This will be one of the examination processes during the code redaction.
- Inconsistent Use Of Exception Handling – in some Java™ programs exception handling is combined with error handling. The use of termination codes makes the exception structure difficult to understand and maintain.

General Guidelines

The exception structure of a program can be improved in several ways. One technique is to apply the methods used for fault–tolerant system in Ada [Litke 90].

- Determine the software compartments – compartmented programs have identifiable boundaries within them that contain the propagation of specific error classes [Litke 90]. Once the compartment has been chosen, an interface to the compartment that includes an exhaustive description of he exceptions that may propagate from it is developed.

  The intent of compartmentalizing the code is to increase its robustness. Robustness is enhanced because the constraints on the system that all exit points from a compartment be specified, including the exception exit points.

  There is no actual definition on what the compartment should be or how it is structured. The compartment can be any set of entities that can raise exceptions. In practice, aligning the compartments with the program structure provides the basis for reasoning about the exception structure.
- Define the interfaces for each compartment – the next step is to determine what exceptions will be allowed to propagate from a compartment. In the majority of integrated application error handling systems, only abstract errors are propagated.

  This is actually a difficult task, since semantically coherent exceptions that describe the complete set of problems that can happen in a compartment need to be defined.
- Limit error handling structure – global error code variables and local exit instructions should be avoided. This guideline will not only ensure simpler structure, but will also facilitate reuse by allowing clients of the components to decide how they should fail.
- Restrict the functional interface – it is easier to implement and verify whether compartmentalization process is working if the interface(s) to the compartments are limited. Any method that is not part of the public interface should be declared `private`.
- Organize exceptions into a hierarchy – for each component, different exceptions can be raised by the same access point. In this way, the client has the option of either recovering from a general component exception or from a particular exception. Care should be taken to ensure that the general exception type chosen as a super–type for the hierarchy is general enough to semantically represent all sub–exceptions.

System Behavior

In order to define the exception handling processes some understanding of the various states of the system and the behaviors that results from those states are described below,

Behavior	Anticipated	Unanticipated
Desirable	Acceptable (normal) behavior *This is a GOOD State*	*This is a BAD State*
Undesirable	Unacceptable (error) behavior *This is an Erroneous State*	Uncontrolled behavior *This is a Disastrous State*

Figure 3 – Behaviors of a less than perfect system

The exception handling aspects of the integrated application must address the following:

- Programs that are preparing to enter a bad state must inform the exception handing system of this process. There is usually not return from the bad state, so the program will likely terminate.
- The number of disastrous states must be minimized through a simple means. This may include:
  - Path testing of each component to verify that all exception conditions addressed. This approach is very time consuming in principle, but for core CORBA component it will be necessary in the end. The path testing process creates a execution path tree and asks the question for each statement on the tree do the possible exceptions on this path have an exception handling process associated with them.
  - Interface expectation exercising in which the interfaces are presented with erroneous data and the resulting execution verified. This is simply good unit testing procedure. This level of testing may or may not have been performed on all the components of the system.

Exception Categories

The reliability of the system is affected by the way exceptions are handled and how consistent the exception handling systems deals with exceptions. Java™ divides exceptions into three categories [Berg 00], [Robillard 99], [Christian 94]:

- Runtime Exceptions – are programming exceptions that should never occur in the lifetime of the system. These are equivalent to the 💣 symbol that displays on an Apple Macintosh. Runtime exceptions are so severe that the system usually terminates.
- Configuration Errors – these are not bugs in the system, but are something that was not set up right. The Exception type is an extension of the base type `Exception`.
- Applications exceptions – are an extension of the type `Exception`, but do not extend from `RuntimeException`. Application exceptions generally

indicate that the program has deviated from its normal flow of control and is trying to back out of what was performed.

In any integrated application, it is unacceptable to crash. This must be prevented by handling all exceptions in some acceptable manner. For exceptions that are not handled, a firewall is needed which catches all uncaught exceptions. This is done at the outer level of the program (`main()` or `run()` is the most likely place) for every thread.

Exception Handling Approaches

There are three (3) approaches to exception handling [Robillard 99a]:

- Termination – assumes the exception is so critical that there is no way to get back to the source of the exception. Whatever component threw the exception decided that there was no way to salvage the situation [Gosling 96]. In the termination mode, the scope of the signaler is destroyed, and if a handled is found, control resumes at the first syntactic unit following this handler.

  Control transfers from the raised point to a handler, terminating all intervening blocks. When the exception handler completes, control flow continues as if the incomplete operation is the protected block terminated without encountering the exception. In this case, the exception handler acts as an alternative operation for the protected code [Gosling 96].
- Resumption – assumes the exception handler is expected to do something to correct the situation, and the normal flow of control will resume as if nothing happened. The faulting method can then be retried, presuming success the second time around. In this case, the exception is more like a method call than an exception handler call. In the resumption model, once an exception is handled, flow of control continues from the point where the exception was originally raised.
- Retry – assumes the exception handler can fix whatever exception occurred and the code retried. In the retry model, the block that signaled the exception will be terminated and retried. For this model to work properly, there must be a clear beginning for the operation to restarted.

  Since the Java language does not naturally support retry, this exception–handling model can be mimicked using a loop and termination model. This may be better than an exception–handling model, since the retry process is explicit rather than hidden. The outcome though is the ability to bypass the exception handling process, so this model should not be used in systems where high reliability is needed.

> This is a fundamental issue in the design of the exception handler – methods should be used to provide resumption–like behavior. This means the program doesn't throw an exception, but calls a method that fixes the problem.

Defining the Exception Classes

At its full maturity, exception classes must be used *system wide*. The common practice is creating customized exceptions is to subclass the `Exception` class. This ensures the compiler checks if it is dealt with properly. Other subclassing strategies can be used if system utilities or hardware related utilities are being managed. In this case, `Error` or `RuntimeException` can be subclassed.

> Do not subclass `Error` or `RuntimeException`, which defeats the whole concept of exception handling.

Since exception classes are full-fledged objects, they have data members and methods. The date members are used to convey information to the local exception handler of the calling client for handling. This information *must* be added to the exception class for each integrated system component for the resulting system to be robust. With the local context information, the throw exception is simply a signal to *stop processing something has gone wrong in the system*. In order for the system to evolve into the robust, survivable application the exception handling methods must be able to correct the exception and return control to the calling method as shown in the resumption and retry model of Figure 4.

The exception classes are created in a hierarchy, so the handler has to option of handling the superclass as a whole, handling the subclasses individually, or handling both classes simultaneously.

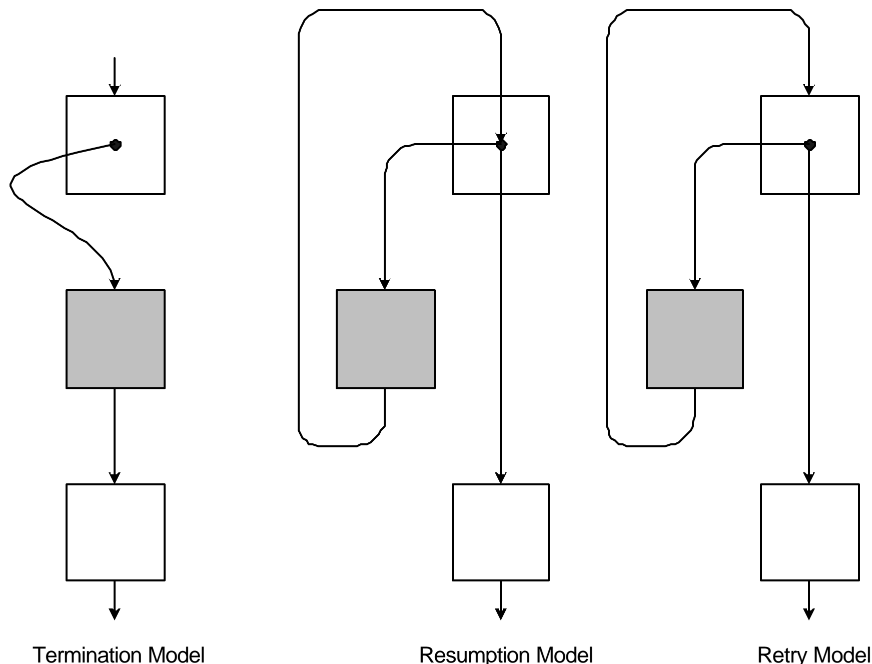

Figure 4 – Three Exception Models

Exception Handling in Java

This section is *restating the obvious*. [1] In java exceptions are represented as objects. As such, they can be instantiated, assigned to variables, passed as parameters, etc. An exception is explicitly signaled using a `throw` statement. Code can be guarded for exceptions within a `try` block. A `try` block is a syntactic scope defining the target. Exception signaled through a `try` block can be caught in one or more catch clauses declared immediately following the `try` block. Optionally, a programmer can provide a `finally` block that is executed independently of what happens in the `try` block. Exceptions thrown in the `finally` block mask any exception that would have been throw in the try block.

`Finally` statements ***MUST*** be used on servers and dedicated services, since they have not clients that can handle the exceptions.

Exception Interfaces

The propagation of exceptions introduces the possibilities of *non–local flow control*. If the caller of a module ignores the exceptions that can cross the module's boundary, the caller cannot adequately prepare for these exceptions. The result is a reduction in the robustness of the system.

The exception *interface* addresses some of these issues, by explicitly specifying exceptions as part of the module's interface. This prevents exceptions not declared in the interface from propagating across the interface.

Exception interfaces create another problem. In practice, these interfaces are not exhaustive. In would be prohibitive to discover and declare the compete set of exceptions that a method or module could raise, mostly because of the high frequency of redundant system exceptions. Since languages that enforce the checking of exception interfaces (Java) typically provide a means of bypassing this checking mechanism. Using the exception hierarchy, Java checks at compile time only for a subset of all exceptions (checked exceptions). The combination of type hierarchy and not checking hat all exceptions have been declared in the exception interface means that developers do not have a precise and complete set of information about the number and types of exceptions potentially crossing the methods boundary.

Users of the Exception Handling System

The Exception Handling system has several users, not all of which are obvious.

- Software – since the real purpose of the exception handler is to keep the system running in the presence of faults, which produce exceptions, which may lead to a failure, the software is the primary user of the exception handler.
- End User – may be another piece of software or an actual human using the software. In this case, the end user will be notified that something has gone wrong in the system and

[1] There are those in the world that object to restating the obvious, but in my experience, it is surprising how the obvious is sometimes not so obvious, especially when dealing with complexity.

some manual intervention is needed to restore the system to a consistent state.

- System Administrator – is the recipient of all the exception log messages.
- Business User – is the ultimate benefactor of the exception handling system, since it maintains the operational consistency of the system.

The key here is that normal exceptions produced by User Input are *not* handled by the Exception Handling system, these are handled by the application through the normal course of the User Interface.

Exception Handling Process

The exception handling process and the code needed to handle exceptions must be built from the bottom up. The following terminology will be used in this paper. This terminology is actually used in all exception handling and fault tolerant systems, but will be stated here for clarity.

- Fault – is the origin of any misbehavior in the system. There are two general types of faults:
  - Design faults, which are actually bugs. These faults need to be reported in a manner that facilities debugging of the system.
  - Service faults, which are usually the absence of a needed resource.
  - Specification faults which are usually invalid inputs to some processing step.

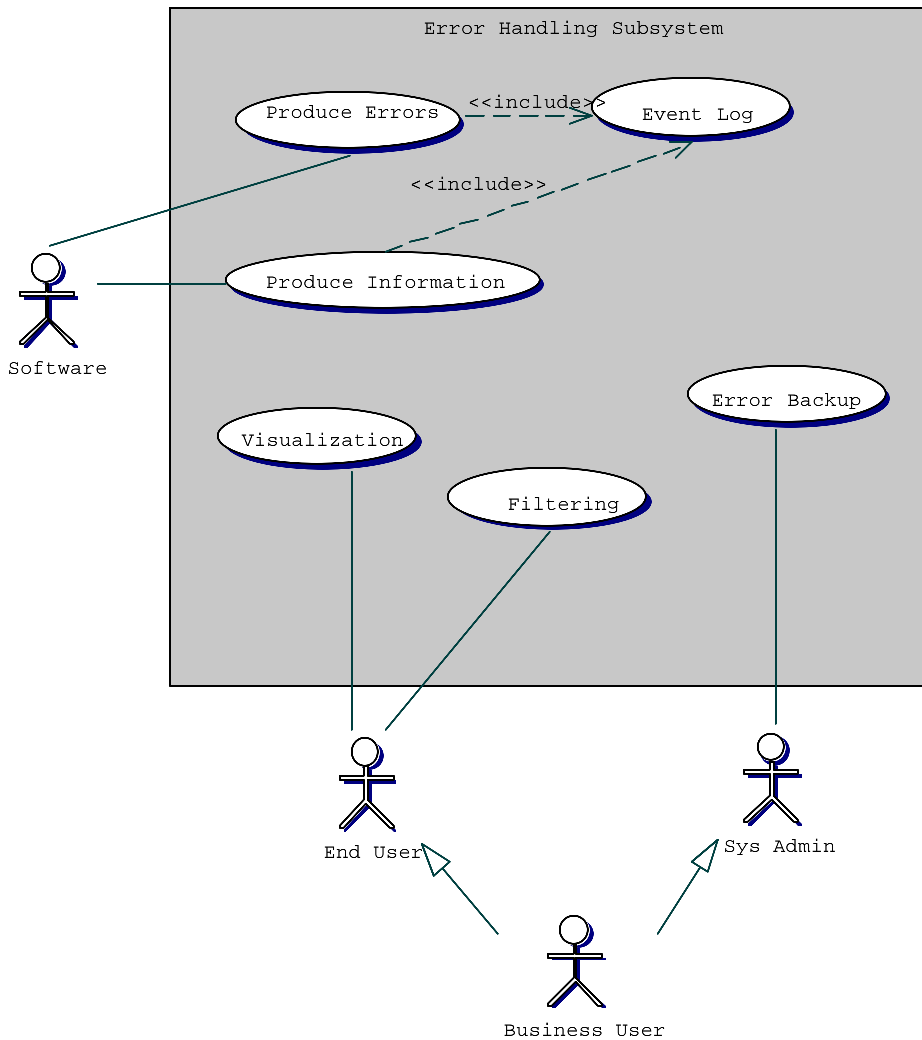

Figure 5 – The Various Participants in the Exception Handling System

- Error – is the manifestation of a fault in the system. There are two general classifications of errors:
  - User Errors – which are mistakes, created by the users when the software is in normal operation. The system is able to react to these errors because it is designed to expect these situations and the error recovery is part of the required functionality of the system. The handling is User Errors should be located in the User Interface and not the Error Handling system.
  - System Errors – which are the result of a fault generated within the system, not resulting from a User Input. These errors are usually associated the absence of a required resources or the unanticipated removal of the facilities provided by the resource.
- Detector – provides and interface between the error and failures happening outside the system and the handling of the errors inside the system. When the Error Handling subsystem is functional, errors and failures of the system are the phenomena that the Error Handling system observes, through the error Detector. There are several types of detector behaviors:
  - Interface checks – which detect errors at the boundaries of interfaces. An invalid parameter indicates that a client of a method did not obey the specified contract for the service. This is often the result of a programming error which can usually only be handled by terminating the program.

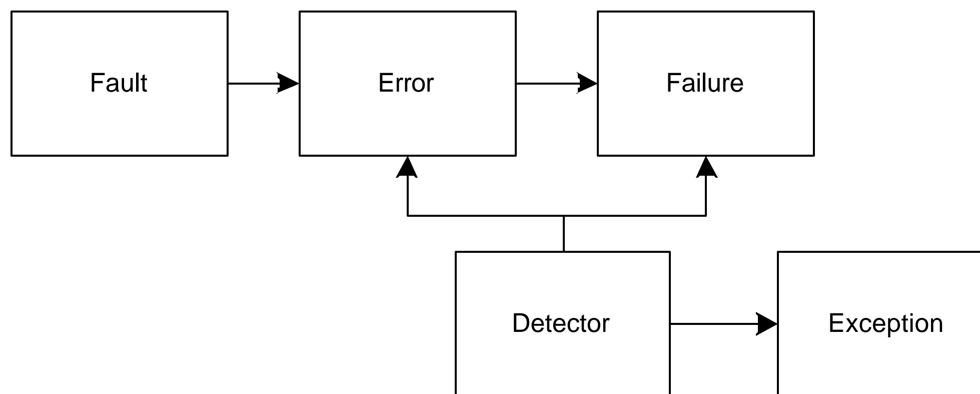

Figure 6 – The Causal Relationships in an exception handling system

  - Constraint checks – are formulated as invariants for the class that contain corresponding data. Invariant should be checked as pre and post conditions of the class operation. Violation of constraints usually results in corrupted data somewhere else in the system, communication failures, or hardware failures.
- Failure – is the deviation of the delivered service from the compliance of the specification. An Error characterizes a particular state of the system, a failure is a particular event of the transition from correct service to incorrect service. A failure is the inability of the software to satisfy it purpose (denial of a service) [Meyer92]
- Exception – is any occurrence of an abnormal condition that causes an interruption in the control flow of the software. An exception is raised (thrown) when a condition is signaled by the software. The response to an exception is to give immediate control to the Exception Handler designated for the exception. This Exception Handler reacts to the exception in a manner designated in the Exception Handler design.

Some General Components of the Exception Handling System

Before proceeding with the detailed design of the Exception Handling system, some general components needed to be defined and described.

- Exception Object – defines the exception information that will be used throughout the exception handling system.
- Exception Logging – defines the mechanism of capturing and logging exceptions.
- Exception Traps – defines which indicators are useful to detect erroneous situations and where these traps should be installed in the source code.
- Exception Handler – where and how the exceptions are handled?
- Exception Dialog – how will the user be alerted to exceptions occurring in the system?
- Resource Manager – how to ensure that the necessary resources are available for the exception handling system to detect and report exceptions?

- Exception Message Abstraction – how can reasonable exception messages be generated without violating the abstraction principles of the system object layers?

Principles of the Exception Handling System

The design of the exception handling system will be based on the following principles:

- User Interaction – in the presence of exceptions the system must behave in a predictable manner. The user should not be presented with confusing and misleading exception messages or exception handling behaviors. The primary problem here is cyclic dependencies in exception handling process. The system detects and exception, reports the exception to the user. The user performs some exception recovery function, which creates the same exception or another exception of similar style.
- Robustness – the exception handling system must be simple. This means that the exception handling code must not have exception states itself. This is usually handled by assuring that all the resources needed for the exception handling code to run are pre–allocated and remain intact during the exception detection and reporting process.
- Separation Of Exception Handling Code – without separating the exception handling code from the mainline application code the overall system can be maintained. The original design of many integrated systems did not address this design pattern, so a redaction must take place to correct this. This can be done under the guise of path tracing to discover the location of the exception detection code.
- Specific Exception Handling Versus Complexity Of Exception Handling – classifying exceptions specifically allows for specific handling procedures. As the system becomes more complex, specific exception names and descriptions also become more complex. By categorizing exceptions into an exception hierarchy, the complexity of the exception messages can be control through abstraction.
- Detailed Exception Information Versus Complexity – whenever an exception occurs suitable information is needed to determine the cause and restore the system to operation. The more complex the system the more complex the exception handling system information needs to be to diagnose the problem. This trade off needs to be carefully managed, since it directly influences the performance and reusability of the exception handling system.
- Performance – the exception handling system should not impose a very big load on the overall system. The measurement of this load as a percentage of system load has not been defined. Typically, this load is not visible in a performance monitor, so it could be said that the exception handing system should impose no load on the system. Since this is not realistic, the load should not add more the 2% to the overall processing load of the integrated application
- Reusability – the services of the exception handling system must be reusable in all environments. The exception handling system is a core component of the system and must be consistent across all platforms as well as application domains.

Addressing the Exception Handling Requirements

At this point in the development cycle, the ideal exception handling system is not available. There are two ways to view this situation:

- The Glass Is Half Empty – the exception handling system must now be added on to the system. Work is now needed to address exception handling that should have been done earlier.
- The Glass Is Half Full – the exception handling system can now be added to the system, since we now know what the system looks like and what kinds of exceptions need to be handled.
- The Glass Is Twice As Big As It Needs To Be – this is the engineer's view of the system and will be the approach taken here.

Exception Handling System Design Process

Since the exception handling system is not fully formed, this is a chance to perform several other activities in conjunction with the creation of a fully formed exception handling and reporting system.

- Visit all the paths of the system to verify that the code can be executed (path analysis tools can be used here). During the visiting process the detection points for exception handling can be examined.
- Normalize the exception message and handling processes by examining each exception detection point.
- Normalize the exception detection and reporting capabilities across the object adapter interfaces interfaces.

There are several advantages and disadvantages in this approach:

- The code needs to be redacted no matter what; so visiting the exception detection point issue will take place anyway.
- Once the system integration process has been stabilized it is time to normalize the exception handling. This usually not be done sooner since the component interface (IDL) will have been changed several times.
- This effort must be done during the redaction process. The old saw of you pay me now or you pay me later. Regarding exception handling we have to pay later – and later has come.

Exception Handling in CORBA Applications

There are several overriding issues in the design of the exception handling system for CORBA based applications.

- The server side applications act as repositories of information and are considered fault–tolerant.
- The client side applications acquire their state information from this repository and can be considered re-startable.

Exception Handling Design Notes

The following *design notes* should be considered during the design and development of programs making use of the exception handling subsystem [Litwak 99], [Govoni 99], [Haggar 98], [Haggar 00], [Warren 99].

Defining Exceptions

- Identify exception classes during the design phase.
- Use exceptions to indicate exceptional and exception conditions, not to return values from methods.
- It is important that code `catch` blocks form the most specific to the most general. This approach will offer the best chance to provide the user with specific, helpful information about what went wrong. Before these messages are emitted, the `catch` blocks should try to correct the exception condition.
- Never declare any variables inside a `try` block if those variables need to be visible outside of it. If this is done, and the variables are accessed an exception will occur.
- Never instantiate an exception object before the time it is actually needed. If this is done the stack trace is printed it will contain misleading information.
- If code that may throw a checked exception is not put inside a `try block`, and do not declare that the method throws an exception, the Java™ complier will not compile the code.
- Because `RuntimeException` exceptions represent problems in the code, they should not be caught and dealt with, but resolved before the code ships. So, don't apply exception handling to runtime exceptions that occur in the Java™ runtime system.
- The older methods of checking return codes are prone to human exception and oversight. Java forces the compiler to catch the class of a caught exception, thereby reducing the margin of exception before program execution.
- Do not use exception for flow control. When this is the case, use `if-then-else` statements. The flow through the code is better expressed with the standard language constructs provided by Java.
- Do not overuse the `catch` clause. If necessary, group similarly related exception classes by a common base class.
- Do not use exceptions for every exception condition. Exception handling was devised as a robust replacement for traditional exception handling techniques found in C++ and Visual Basic. This one of the major contributions of Java, but it can be overused. Use exceptions for conditions outside the excepted behavior of the code.
- Use exceptions for conditions outside that of expected behavior. Exceptions should most often occur when improper states arise in the system that must be dealt with in a strict manner.
- Always extend the `Exception` class. Doing so guarantees proper operability within Java and between Java API's and possibly other frameworks or libraries.
- Know the mechanics of exception control flow. In order to handle exception properly it is necessary to have a firm understanding of both the application flow and the exceptions the application can produce.
- Consider the drawbacks to the `throws` clause. Adding an exception to a `throws` clause affects every method that calls it.

Handling Exceptions

- Regardless of the type of exception being thrown, when an exception is thrown, any statements after the throw point will not be executed, because the control is given back to the invoking context.
- Use exception for all exception conditions. If the conditions are simple, exception codes may prove less wieldy. When exceptions that are more complex are handled then there will be mixture of exception handling techniques, undoing all the simplicity of the first approach.
- Propagate exceptions between interacting frameworks or subsystems. This will allow conditions occurring at a low level with one system to be mapped cleanly to higher–level condition in the integrated system.
- Attempt to restore objects to their proper functioning state after receiving an exception. Even if the design intends to pass an exception along to a higher–level method, ensure that an given object receiving to throwing an exception is not left in a negative state, where it might be reused and compromise the integrity of the system. This can be done by leaving the object in the state it was in before the exception occurred. This may involve using the `finally` clause to restore the object to a consistent state.
- Return object to a valid state before throwing an exception. Proper exception handling only begins when the exception is thrown. The explicit purpose of an exception is to notify someone or something that the exception occurred. There is an implicit purpose though – to allow the system to recover and continue to run in the presence of the exception.

- Never ignore an exception. When an exception is generated, it must be caught. If it is not caught, the thread of execution terminates.
- Never hide an exception. Only the last exception generated I propagated upward. The original cause of the expectation may be hidden. This is a problem when the code is attempting to recover from one exception and another exception occurs.
- Be specific and comprehensive in the `throws` clause. When specifying the throws clause, fill it out completely. Although the compiler does not enforce this, it is good programming practice.
- Use `finally` to avoid resource leaks. Using `finally` enables the code to execute whether or not an exception occurred or not.
- Do not return from a `try` block.
- Place `try-catch` blocks outside of loops. Exceptions can have a negative impact on the performance of the code.
- Throw exceptions from constructors. Constructors cannot return an exception code, so throwing exceptions is one way to inform someone that the constructer failed.
- Catch as many exceptions as possible explicitly – avoid `catch(Exception)` as the only exception handler.
- Avoid using `try (…) catch(…)` on a per method basis for all methods within a block.
- Separate fatal and non–fatal exception class hierarchies.
- Reduce the overall number of exception classes by categorizing them and using a constant (typesafe) to represent the condition.
- Understand the implications of throwing exceptions in constructors.

Handle or Declare an Exception?

How can it be decided whether to use `try-catch` (handle the exception) or declare the method throws and exception? It is best to use the `try-catch` because the method is in the best position to know what to do with the exception. There are two alternatives here with different selection criteria:

- The `try-catch` approach is best used for application level code that can discover the context of the exception in provide a direct solution within this context
- If the method is a low level piece of code, exceptions should be thrown to the caller, since the low level method may not have the context in which to resolve the exception.

A Simple Exception Handling Architecture

> We are ready for any unforeseen event that may or may not occur.
>
> — Dan Quayle

The following diagram illustrates the *simple* exception handling architecture for a COTS–based application. This architecture will be expanded in more detail later, but for now this is the *pattern* that will be used in all domains for performing error handling.

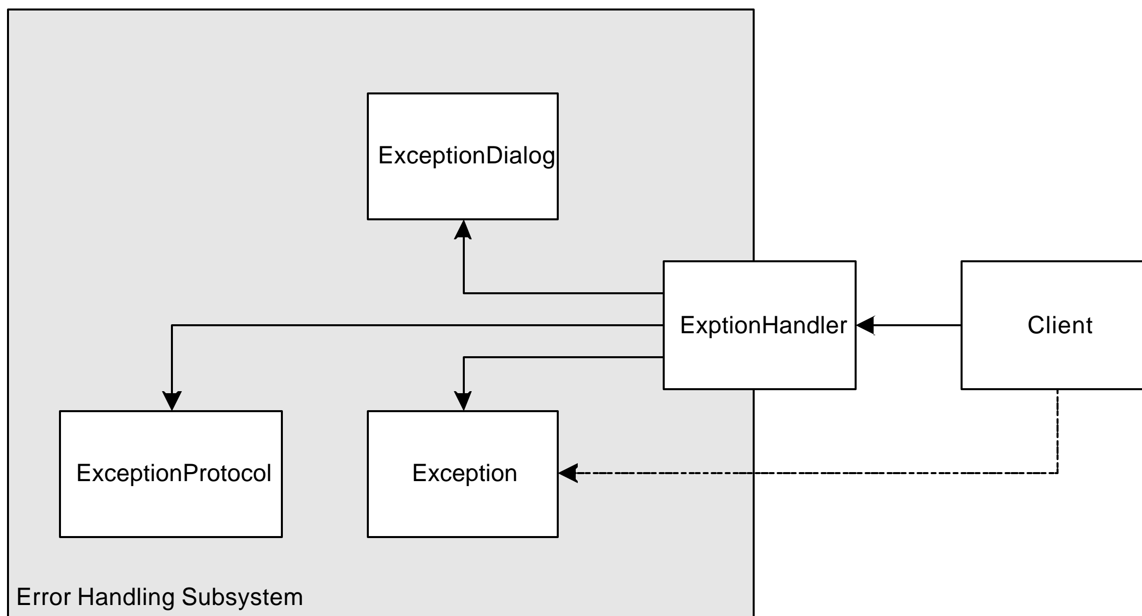

Figure 7 – The Structure of the Exception Handling Subsystem

Components of the Exception Handling Subsystem

The following components are used to build the exception handling subsystem:

- Exception – encapsulates an `ExceptionObject` which contains all the information about the exception that needs to be reported.
- ExceptionProtocol – is a class responsible for writing the exception to a log file. It records the details of the exception including the calling chain, the location of the exception.
- ExceptionDialog – is a class responsible for communicating with the user of a user–agent in the presence of an exception.
- ExceptionHandler – encapsulates the exception handling process. This class helps enforce a consistent form and function for the exception handling process.
- Client – once an exception has occurred in the application the client creates and `ErrorObject` and adds the necessary information needed to report and handle the exception. This object is then passed to the `ExceptionHandler` for processing.

A Simple Sequence for Exception Handling

The actual implementation of the Exception Handling system involves the *injection* of exception handling code in a variety of places in the application. Before proceeding with the scope of work a *simple* exception handling protocol is presented.

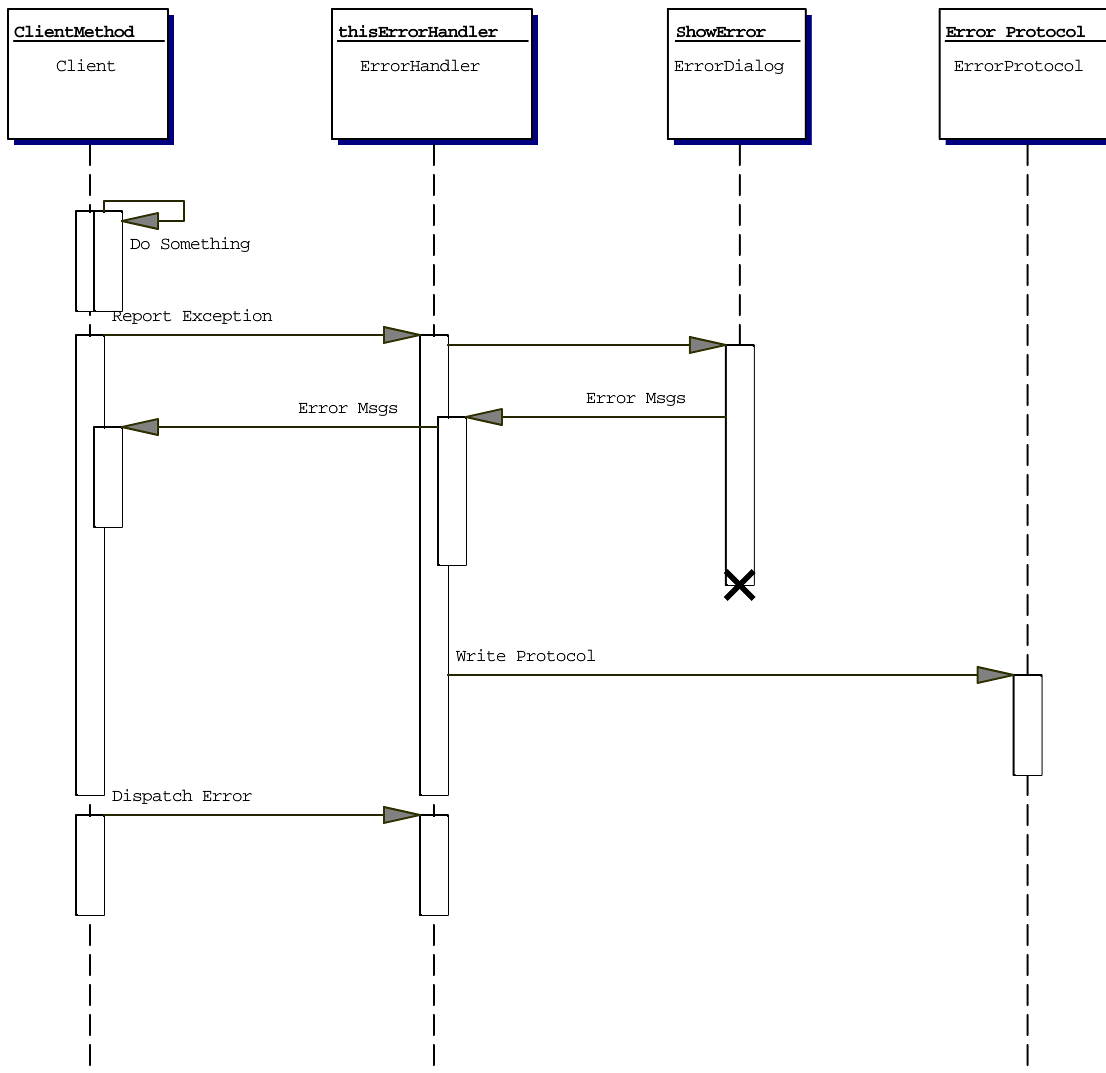

Figure 8 – Simple Exception Handling Sequence

The Big Picture

The deployment of a simple error or exception handling scheme is necessary but not sufficient to produce robust software applications. A more complex is required to handle both the static and dynamic exceptions that occur ion modern object oriented systems.

Figure 9 describes the overall design of the Exception Handling system. Each class is described below in some level of detail.

- ExceptionFactorySelector – provides the foundation classes for the ExceptionFactory Manager. This allows multiple exception factories to be created for the various domains of the system. Since there are distinct programming language, operating system, database, and application environments, threes can be isolated through a multi–layer factory scheme.
- ExceptionFactoryManager – creates an exception factory for each domain.
- ExceptionFactory – is the abstract exception factory for the exception handling system. This factory makes exception objects that are used to convey the exceptions occurring in the applications. This class makes use of the Abstract Factory pattern found in most patterns texts.
- ConcreteExceptionFactory – this is the actual factory class that serves as the traffic cop for the exception handling class generation. In many object oriented designs a single class decides which sub–classes to instantiate. In the Factory Pattern, the super–class defers the decision to each sub–class. The decision point is actually made in the abstract class (ExceptionFactory) that creates objects but lets leach sub–class decide which object to create.
- ExceptionType – is an abstract class factory for making the actual exception classes
- ConcreteException – the exception class that is included in the `ExceptionClassInstance` by the exception factory. This object is instantiated when the exception occurs and represents the application and domain specific information about the exception.
- ExceptionClassInstance – a specific instant of the exception object. Since this object will be forwarded to other domains, there must be specific identifiable instances of the object that can be passed across CORBA boundaries.
- ExceptionProtocol – is a singleton that is responsible for writing exception information to a log file. This log file may be located in the local environment or may be located across the network on another server. Access to the log file is provided through another object that hides the physical location of the file, its writing protocol, and the methods of locating the file. The records written to this file include the dynamic call chain, which reflects the control flow up to the point where the exception was detected.
- ExceptionHandler – is a singleton for the exception handling subsystem. The ExceptionHandler encapsulates the exception processing and enforces the consistency of the exception handling strategies. The client uses the exception handler to process any exceptions generated in the application. The ExceptionHandler uses the ExceptionDialog to display exceptions that the user can correct. The ExceptionHandler uses the ExceptionProtocol to generate exception objects and process them.
- DefaulltExceptionHandler – this is the default code components used by the `catch` clause in the code. This code will be external to the application code, usually in a helper class. The Exception Handler is separate from the application exception handling so common changes to the exception handling can be made in the helper class, without having to touch each application code segment. This provides a flexible technique for isolating exception codes, language impacts and distributed processing impacts.
- GeneralClientClasses – this is a placeholder of the client application.
- ExceptionDetector – this is a generic placeholder for the exception detection code. The actual exception detection logic is usually application specific but there are general guidelines for detecting and handling exceptions that can be found in any good software engineering resource.
- Client – this is the client of all this exception handling code. The client can be a Java or C++ application with `catch` clauses for exception handling. There are detailed rules for writing robust software in these languages, and the system analysis and code review should address these issues.
- ExceptionResource – this set of classes provides pre–allocated resources for the exception handling services. When an exception occurs it may be because resources are running
- ResourceAlloc – the pre–allocation of resources provides the opportunity for the exception handling subsystem to continue to operate in the presence of out of memory situations. This is most common in C++ code. In most cases, the resources needed include string space, file handles, object space, and other dynamic resources that are allocated during runtime.

  In many instances, memory leaks occur during the normal operation of the application. Pre–allocating resources allows the exception handling system to continue (for some time) in the presence of these memory leaks.

  There are many resources can be pre–allocated:

  - Log file connections
  - Printing connections
  - User interface dialogs

- Shut down space for entities that must be saved to memory before being moved to a persistent location

Here's an example of a C++ class that is protected from out–of–memory errors:

```
const int NumberOfPreallocatedExceptions = 200;
const int ExClassIdLength = 80;
…
// an exception class that is safe against out-of-
memory errors
class ExBaseSafe {
public:
      void * operator new(size_t size);
      void operator delete(void* deadObject, size_t
size);
      virtual ~ExBaseSafe( void );
      ExBaseSafe (
        const char pszExClass[ExClassIdLength],
        const char pszExNumber[ExNumberIdLength],
        const char pszExText[ExTextIdLength] =
"undefined");

      ExBaseSafe ( const ExBaseSafe & eExceo );

// the list of functions has been shortened

private:
      // some things must be forbidden
      ExBaseSafe & operator = ( const ExBaseSafe &);
      ExBaseSafe (){};
      // organizational variables for memory management
      BOOL ifIsUsed;
      static int iFreePrototypeIndex;
      static ExBaseSafe
iPrototype[NumbersOfPreallocatedExceptions];
      // normal instance variables must be fixed length
      char iszExClass[ExClassIdLength];
        // unique identifier of an exception
…
};
```

- ConcreteExceptionDetector – this is the placeholder for the actual code fragments of the exception detector.
- ExceptionDialog – some form is exception dialog is needed for interactive applications where a User Interface is provided. These classes provide this service.
- WinDiagLogger – is responsible for writing exceptions to a window on the users workstation. This dialog is generally modal in that the dialog is presented to the user and the user alters something and the dialog responds to an acknowledgement that the action took place or another exception occurs. This dialog will make use of low–level facilities in the native operating system and not depend on the application itself for presenting the dialog.
- FileDiagLogger – is responsible for logging exceptions to a file system for later analysis.
- Exceptions – is an abstract base class for all exceptions in the system
- RunTimeException – one of the exception types dealing with run time problems in the system.
- InterfaceException – an exception in the user interface or some other system interface.
- DomainException – an exception in the application domain
- EnvironmentException – and exception in the application environment.
- Resource Exception – a resource exception.
- InfrStructException – an infrastructure exception
- CorbaException – a CORBA exception.
- ExceptionInterface – an abstract interface to the exception classes

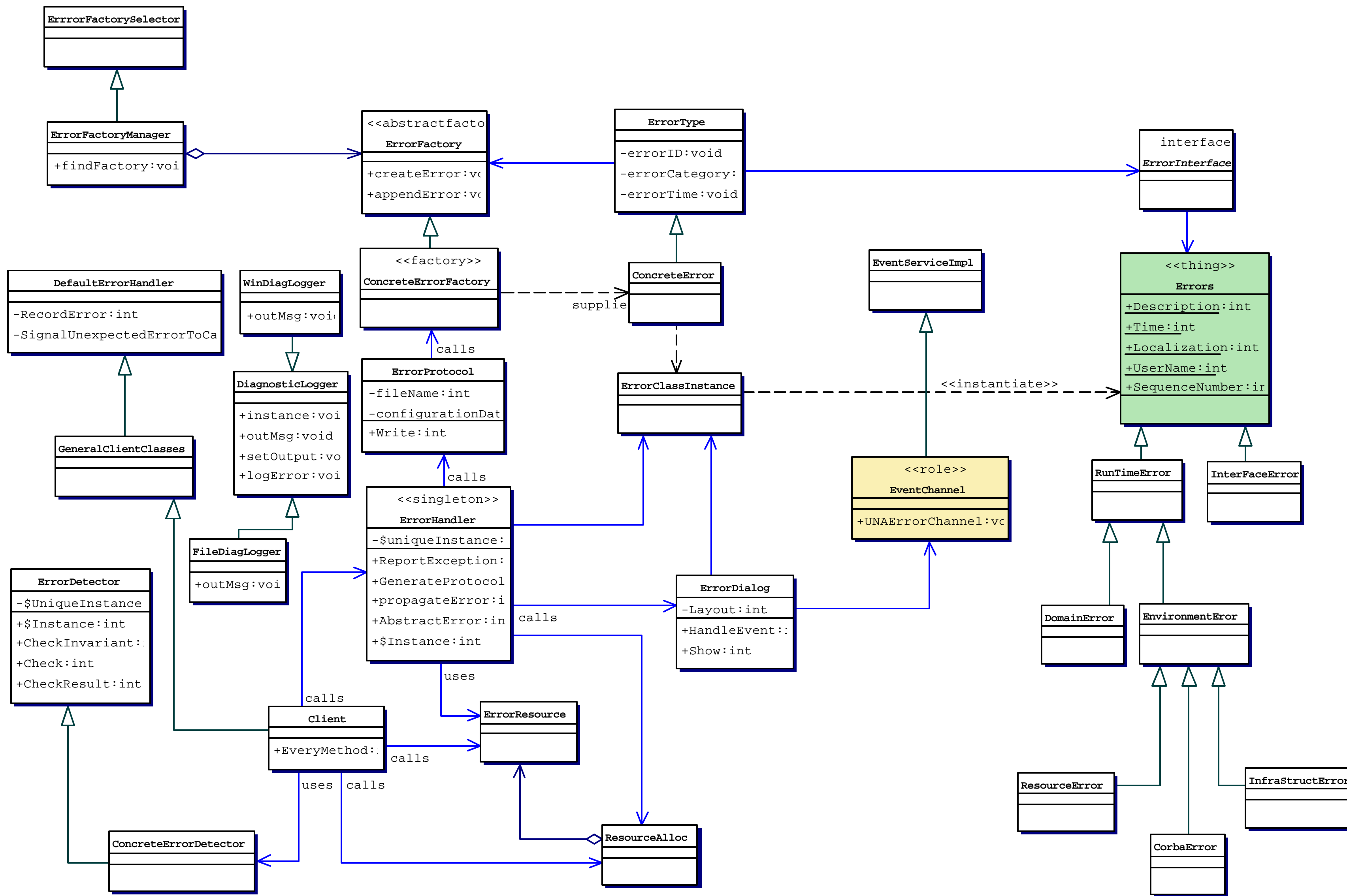

Figure 9 – Master Class Diagram for the Exception Handling Subsystem

Exception Handling Within CORBA

The detection and handling of exceptions within a Java Method, Class, and Application is a relatively straightforward task. Handling exceptions within the architecture of an integrated application using the facilities of CORBA is a more complex process. Exceptions generated by a call to a component method must be propagated back to the caller through the exception system. An unhandled exception must be propagated back to the caller even when the caller is remote, or when the exception itself is generated from a remote call.

CORBA provides an exception hierarchy similar to Java. CORBA defines two types of exceptions:

- System Exceptions – which are standard exceptions that are thrown by CORBA.
- User–Defined Exception – which are defined in the IDL as a structure that contains data fields.

In the Java deployment of CORBA, the CORBA exceptions are derived from `java.lang.RuntimeException`. These exceptions are mapped to Java classes that extend `org.omg.CORBA.SystemException`. These exception classes proved access to the major and minor exception codes of IDL as well as the string describing the reason for the exception. The Java class name for each standard IDL exception is the same as its IDL name as declared in the `org.omg.CORBA` package.

Here's an example:

```
Exception NotInterested {string explanation;};
final public class NotInterested extends
org.omg.CORBA.UserExpection
{
      public java.lnag.String exception;
      public NotInterested () {}
      public NotInterested(java.lang.String,
explanation)
      {
        this.explanation = explanation;
      }
      public java.lang.String toString()
      {
        org.omg.CORBA.Any any =
org.omg.CORBA.ORB.int().create_any();
        NotInterestedHelper.insert(any, this);
        Return any.toString();
      }
}
```

Reporting Exceptions Across CORBA Links

The generation of exceptions will that place in much the same manner as Java applications within a single domain. The handling of these exceptions becomes a different issue though. The domain knowledge of the exception is intentionally hidden from the user through a CORBA object interface.

References

> Everywhere I go I'm asked if I think the university stifles writers. My opinion is that they don't stifle enough of them.

The following reference materials were used as the basis for this design. In the patterns world, the first activity of the design process is to search for a pattern that somehow meets or clarifies the design requirements. The domain of exception handling is not very interesting to the normal development community, as exhibited by the late introduction of this design into a COTS–based project.

I would encourage anyone reading this document to understand the complexity as well as the *beauty* of robust systems and the underlying software that make them so.

- [Ahronovitz 00] "Exceptions in Object Modeling: Questions from an educational experience," Yolande Ahronovitz, Marianne Huchard, *ECOOP 2000*, June 12, 2000.
- [Belady 76] "A Model of Large Program Development," L. A. Belady and M. M. Lehman, *IBM Systems Journal*, 15(3), pp. 225–252, 1976.
- [Berg 00] *Advanced Java™,* Clifford J. berg, Prentice Hall, 2000.
- [Bertino 00] "Do Triggers Have Anything To Do With Exceptions?" Elisa Bertino, Giovanna Guerrini, Isabella Merlo, *ECOOP 2000*, June 12, 2000.
- [Borror 99] *Java™ Principles of Object–Oriented Programming*, Jeffry A. Borror, R&D Books, 1999.
- [Buhr 00] "Advanced Exception Handling Mechanisms," Peter Buhr amd W. Y. Russel Mok, *IEEE Transactions on Software Engineering*, 26(9), September, 2000.
- [Christian 89] "Exception Handling," Flaviu Christian, in Dependability of Resilient Computers, T. Anderson Editor, BSP Professional Books, 1989, pp. 68-97.
- [Christian 95] "Exception Handling and Tolerance of Software Faults," F. Christian, in Software Fault Tolerance, edited by M. R. Lyu, John Wiley & Sons, 1995, pp. 81–107.
- [Coplien 95] "Fault–Tolerant Telecommunications System Patterns," James Coplien, Michael Adams, Robert Gamoke, Robert Hammer, Fred Keeve and Keith Nicodemus, AT&T Bell Laboratories, 1995.
- [Cunningham 95] "The CHECKS Pattern Language for Information Integrity," Ward Cunningham, in *Pattern Language of Program Design: Volume 1*, edited by James Coplien and Douglas Schmidt, pp. 145–155, Addison Wesley, 1995.

- [Dellarocas 98] “Toward Exception Handling Infrastructure for Component–Based Software,” Chrysanthos Dellarocas, A position paper presented at the *1998 International Workshop on Component–Based Software Engineering*.
- [Défago 97] “Reliability with CORBA Event Channels,” X. Défago, P. Felber, B. Garbinato, and R. Guerraoui. In *Proceedings of the 3rd Conference on Object–Oriented Technologies and Systems (COOTS),* Portland (Oregon), June 1997. http://lsewww.epfl.ch/OGS/publications.html#CORBA.
- [Eckel 98] *Thinking in Java™,* Bruce Eckel, Prentice Hall, 1998.
- [Fahndrich 98] “Tracking Down Exceptions in Standard ML Programs,” M. Fahndrich, J. Foster, J. Cu, and A. Aiken, *Technical Report CSD–98–996*, University of California, Berkley, February, 1998.
- [Felber 98] “The CORBA Object Group Service A Service Approach to Object Groups in CORBA THÈSE N° 1867,” Pascal Felber, École Polytechnique Fédérale De Lausanne, http://lsewww.epfl.ch/OGS/thesis/, 1998
- [Garcia 00] “An Exception Handling Software Architecture for Developing Robust Software,” Alessandro F. Garcia, Cecilia M. F. Rubira, *ECOOP 2000*, June 12, 2000.
- [Goodenough 75] “Exception Handling: Issues and A Proposed Notation,” John Goodenough, Communications of the ACM, December 1975.
- [Gosling 96] *The Java Language Specification*, J. Gosling, Addison Wesley, 1996.
- [Govoni 99] *Java™ Application Frameworks*, Darren Govoni, John Wiley & Sons, 1999.
- [Haggar 98] “Java Exception handling,” Lecture at Software Development East, ’98, Washington DC, 1998.
- [Haggar 00] *Practical Java™: Programming Language Guide*, Peter Haggar, Addison Wesley, 2000.
- [Hamlet 99] “Theory of System Reliability based on Components,” Dick Hamlet, Dave Mason, Denise Woit, *ISSRE ’99*, Boca Raton, FL, November 1999.
- [Hammer 98] “Telecommunications Input and Output Pattern Language,” Robert Hammer and Greg Stymfal, Lucent Technologies and AG Communications Systems, 1998.
- [Hansen 00] “Adapting C++ Exception Handling to an Extended COM Exception Model,” Bjorn Egil Hansen, Henrik Fredholm, *ECOOP 2000*, June 12, 2000.
- [Harrison 98] “Patterns for Logging Diagnostics Message,” Neil B. Harrison, in *Pattern Language of Program Design Volume 3*, edited by Martin Riehle and Frank Buschmann, pp. 277–289, Addison Wesley, 1998.
- [Hissam 98] “Isolating Faults in Complex COTS–Based Systems,” Scott Hissam and David Carney, *SEI Monographs on the Use of Commercial Software in Government Systems*, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, February 1998.
- [Knudsen 00] “Exception Handling versus Fault Tolerance,” Jorgen Lindskov Knudsen, *ECOOP 2000*, June 12, 2000.
- [Kienzle 00] “Exception Handling in Open Multithreaded Transactions,” Joerg Kienzle, *ECOOP 2000*, June 12, 2000.
- [Litke 90] “A Systematic Approach for Implementing Fault Tolerant Software Designs in Ada,” J. D. Litke, *Proceedings of the Conference on TRI–ADA ’90*, pp. 430–408, ACM, December 1990.
- [Litwak 99] Pure Java™ 2, Kenneth Litwak, Sams, 1999.
- [Lopes 00] “Using AspectJ™ For Programming The Detection and Handling of Exceptions,” Cristina Lopes, Jim Hugunin, Mik Kersten, Martin Lippert, Erik Hilsdale, Gregor Kiczales, *ECOOP 2000*, June 12, 2000.
- [Maffeis 95] “Adding Group Communication and Fault-Tolerance to CORBA,” Silvano Maffeis, *USENIX Conference on Object-Oriented Technologies (COOTS)*, June 26-29, 1995, pp. 135-146
- [Meyer 92] *Effective C++,* Scott Meyer, Addison Wesley, 1992.
- [Mikhailova 00] “Behavior-Preserving Evolution of Interface Exceptions,” Anna Mikhailova, Alexander Romanovsky, ECOOP 2000, June 12, 2000.
- [Orfali 98] *Client / Server Programming with Java and CORBA: 2nd Edition*, Robert Orfali and Dan Harkey, John Wiley & Sons, 1998.
- [Parnas 72] “On The Criteria To Be Used In Decomposing Systems Into Modules,” David L. Parnas, *Communications of the ACM*, 15(12), December 1972, pp. 54–60.
- [Perry 00] “Current Trends on Exception Handling,” D. E. Perry, A. Romanovsky, and A. Tripathi, IEEE Transactions on Software Engineering, 26(9), September 2000.
- [Pessaux 99] “Type–based Analysis of Uncaught Exceptions,” F. Pessaux and X. Leroy, *Proceedings of the 26th Symposium on the Principles of Programming Languages*, pp. 276–290, January 1999.
- [Patino-Martinez 00] “Exception Handling in Transactional Object Groups,” Marta Patino-Martinez, Ricardo Jimenez-Peris, Sergio Arevalo, *ECOOP 2000*, June 12, 2000.
- [Renzel 94] *Error Handling for Business Information Systems: A Pattern Language*, Klaus Renzel, www.sdm.de.g.arcus/cookbook. This reference may or may not be at the specified location, since sd&m is a commercial site and has moved things in the past.
- [Robillard 99] “Analyzing Exception Flow in Java™,” M. P. Robillard and G. C. Murphy, *Proceedings of the Joint 7th European Software Engineering Conference and the 7th ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, Volume 1687 of

Lecture Notes in Computer Science pp. 322–337, Springer–Verlag, September 1999. Also at www.computer.org for IEEE CS members.

- [Robillard 99a] "Analyzing Exception Flow in Java™ Programs," Martin P. Robillard, Masters Thesis, University of British Columbia, September 1999.
- [Robillard 99b] "Regaining Control of Exception Handling," Martin P. Robillard and Gail C. Murphy, Technical Report Number TR–99–14, University of British Columbia, December 1st, 1999.
- [Romanovsky 99] "An Exception Handling Framework for N–Version Programming in Object Oriented Systems," Alexander Romanovsky, *Technical Report CS–TR–684*, University of Newcastle, November 1999,
- [Romanovsky 00] "Exception Handling in Cooperative and Competitive Concurrent Object-Oriented Systems," Alexander Romanovsky, Joerg Kienzle, ECOOP 2000, June 12, 2000.
- [Sebesta 96] *Concepts of Programming Languages*, R. W. Sebesta, Addison Wesley, 1996.
- [Sinha 99] "Criteria for Testing Exception Handling Constructs in Java™ Programs," S. Sinha and M. Harrold, *Technical Report OSU–CISRC–6/99–TR16*, The Ohio State University, June 1999. Publishing in *Proceedings of the International Conference on Software Maintenance*, August 1999.
- [Sinha 98] "Control-Flow Analysis of Programs with Exception-Handling Constructs," Saurabh Sinha and Mary Jean Harrold. *Technical Report OSU-CISRC-7/98-TR25*, Department of Computer and Information Science, The Ohio State University, July 1998.
- [Sinha 98a] "Analysis of Programs with Exception-Handling Constructs," Saurabh Sinha and Mary Jean Harrold, *Proceedings of the International Conference on Software Maintenance*, pp. 348–357, Bethesda, MD, November 1998. Also in *IEEE Transactions on Software Engineering*, 26(9), September 2000, pp. 849–871.
- [Stevens] "The Effect of Java Exceptions on Code Optimisations," Andrew Stevens, Des Watson, ECOOP 2000, June 12, 2000.
- [Tripathi 00] "An Exception Handling Model for a Mobile Agent System," Anand Tripathi, Robert Miller, ECOOP 2000, June 12, 2000.
- [Valkevych 00] "Formalizing Java Exceptions," Tatyana Valkevych, Sophia Drossopoulou, *ECOOP 2000*, June 12, 2000.
- [Vo 98] "Xept: A Software Instrumentation Method for Exception Handling," Kiem–Phong Vo, P. Emerald Chung, and Yennun Huang, Lucent Technologies.
- [Warren 99] *Java in Practice*, Nigel Warren and Philip Bishop, Addison Wesley, 1999
- [Xu 98] "Coordinated Exception Handling is Distributed Object Systems: From Model to the System Implementation," Jie Xu, Alexander Romanovsky, and Brian Randell, Proceedings of the 18th International Conference on Distributed Computing, IEEE, 1998.
- [Yi 98] "An Abstract Interpretation for Estimating Uncaught Exceptions in Standard ML Programs," K. Yi, Science of Computer Programming, 31:147–173, 1998.
- [Yi 99] "Exception Analysis for Java™," K. Yi, ECOOP'99 Workshop on Formal Techniques for Java™ Programs, June 1999. This is a difficult paper to find at the ECOOP '99 site, but it is located at http://ropas.kaist.ac.kr/~kwang/publist.html.
- [Zilles 98] "The Use of Multithreading for Exception Handling," Craig B. Zilles and Gurindar S. Sohi, Proceedings of the 32nd Annual International Symposium on Microarchitecures, 1998.
- [Zinky 97] "Architectural Support for Quality of Service for CORBA Objects," John A. Zinky, David E. Bakken, and Richard D. Schamntz, *Theory and Practice of Object Systems*, 3(1), 1997.

Niwot Ridge Consulting
4347 Pebble Beach Drive
Niwot, Colorado 80503
www.niwotridge.com
720.406.9164

In Association with

Nilan–Sanders Associates
1792 Kettle
Littleton, Colorado 80120
303.798.9741

Revision 3.3 – 10/13/2000
Printed 4/24/01 1:51 PM