



Error Handling for Business Information Systems

A Pattern Language

Version 1.1

Klaus Renzel

sd&m München

27/08/2003

1 INTRODUCTION	1
1.1 Motivation	2
1.1.1 Why Error Handling?	2
1.1.2 Fault Tolerance	3
1.1.3 What to do?	4
1.1.4 How to do it?	6
1.2 Terminology	6
1.2.1 Fault	7
1.2.2 Error	7
1.2.3 Failure	8
1.2.4 Detector	8
1.2.5 Exception	9
1.2.6 How are these terms related to each other?	10
1.3 Pattern Roadmap	12
1.3.1 Pattern Catalog	14
1.4 Notational Conventions	16
2 PATTERN LANGUAGE	17
2.1 Architecture	18
Error Handling Framework	19
2.2 Error Types and Structure	30
Error Object	31
Exception Hierarchy	37
2.3 Error Detection	41
Error Traps	42
Assertion Checking Object	50
2.4 Error Logging	54
Backtrace	55
Centralized Error Logging	61
2.5 Error Handling Strategies	64
Default Error Handling	65
Error Dialog	68
Checkpoint Restart	76
Error Handler	80
Resource Preallocation	84
2.6 Integration	89
Exception Abstraction	90
Exception Wrapper	93
2.7 Multithreading	97
Multithread Exception Handling	98

3 DESIGN RULES	102
APPENDIX A GLOSSARY	107
APPENDIX B CHECKLIST	110
APPENDIX C SOME THEORY	113
APPENDIX D THE ARIANE 5 FAILURE	123
REFERENCES	127

1

Introduction

This paper is about the design and implementation of error handling facilities in business information systems. It presents a pattern language for the object-oriented design of components needed for error handling and also present sample code in different programming languages (C++, Java, Cobol, Smalltalk).

The pattern language does not contain an ultimate design for error handling. Some patterns result from pattern mining activities among various sd&m projects whereas others reflect own ideas or input from other sources. Thus, it is a recording of a current status. Hopefully, the document will mature by your feedback: criticism, suggestions for improvement, known uses which are yet undocumented or even new patterns which should be included. In short, all kinds of comments are welcome.

1.1 Motivation

1.1.1 Why Error Handling?

There are two ways of producing error-free software. But only the third will work ...
[Unknown author]

Reliability is a major characteristic of high-quality software. Software should behave well in nearly every situation. During the development process we try to avoid, detect and remove as many errors as possible.

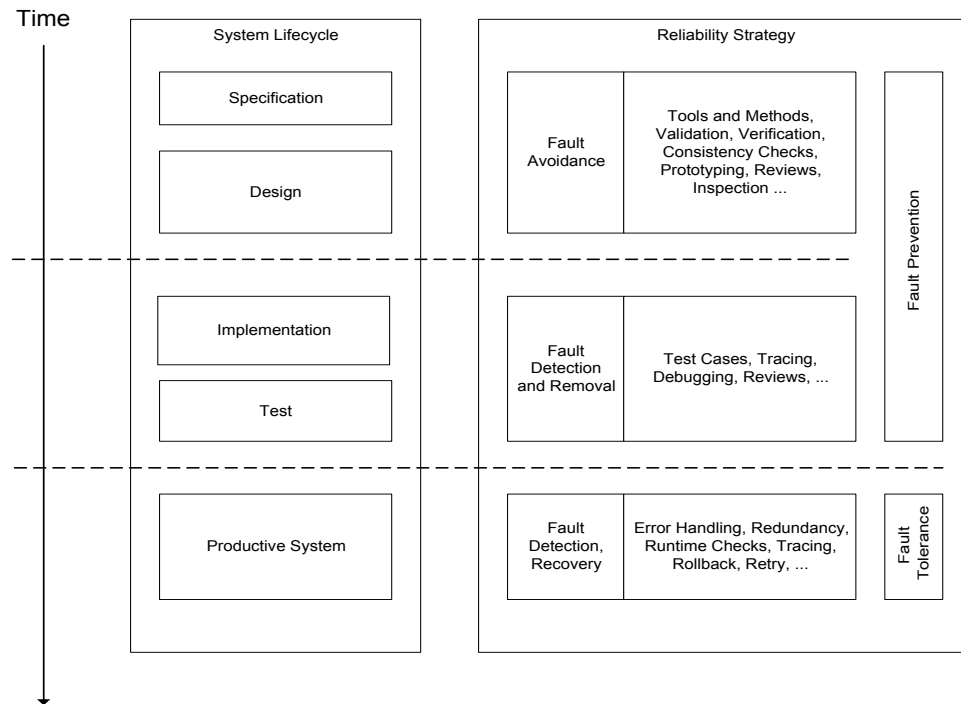


Figure 1: Approaches to reliability

It is our goal to develop complete specifications (no input states with unspecified behaviour) and correct implementations.

Nevertheless, we cannot assume that the delivered software is free of errors. Therefore, we have to think about mechanisms to protect the productive system from unacceptable behaviour while erroneous situations occur (see Figure 1).

Those mechanisms become a part of the basic infrastructure, which is used by nearly every part of the software. Thus the structure of the whole software is extremely influenced by the error-handling mechanism and therefore has to be designed carefully. It is not an add-on feature we can think of during the implementation.

Error handling is an important part of the software architecture and thus has to be considered of during specification and design of an application!

Figure 1 (adapted from [LA90]) summarizes the different approaches to reliability and relates them to phases of the software lifecycle. Up to delivery the strategy is fault prevention: First, the development team concentrates on avoidance of any errors, and secondly, during the test phases (component and integration tests) the team tries to detect and eliminate existing errors. Once the product is delivered the strategy is fault-tolerance: the software must be able to detect errors and to recover from them online. But bugs cannot be fixed on-line, so developers must do that off-line as part of the maintenance phase which accompanies the introduction of a new software system at the customers site (not shown in Figure 1).

1.1.2 Fault Tolerance

The major difference between a thing that might go wrong and a thing that cannot possibly go wrong is that when a thing that cannot possibly go wrong goes wrong it usually turns out to be impossible to get at or repair.

Douglas Adams [Ada92]

A system able to recover from errors and to restore normal operation is called fault tolerant. Depending on the criticality of a system fault tolerance has many facets. Especially in safety critical applications (for instance the control systems for space shuttles, nuclear power plants) fault tolerance is a very important topic (see Appendix Appendix D) and a variety of mechanisms is used to achieve this tolerance (n-version programming, backups, replication). In the domain of business information systems fault tolerance plays a different role: the major concern is to guarantee the correctness and integrity of the data and to prevent any corruption and loss of data. In an error situation (especially design faults) it is generally not possible to correct the fault on-line and so it is probable that the same error occurs again. Furthermore, there is the danger to proceed with inconsistent or corrupted data. Thus in most cases we do not try to recover (only if sensible) from errors in order to continue normal operation, but we try to terminate (when necessary and possible) the application in a consistent way. This failfast approach is a „light“ form of fault tolerance and is merely directed to develop a robust system, that is a system with a predictable behaviour in nearly every situation.

We can summarize the essence of this section to the following rule of thumb:



Error recovery in fault tolerant systems is very complex and expensive. It is used in safety critical applications where availability and timing are important requirements. In the domain of business information systems, which are often critical too but with other requirements, it is usually more effective to invest in error prevention during development and robustness of the productive system.

This does not mean that information systems are less critical or uncritical (on the contrary e.g. the bank and insurance business heavily rely on their information systems), but the safety strategy and mechanisms are primarily based on technical infrastructure, backups and redun-

dant hard- and software. So safety aspects (e.g. a fail-safe environment) are not solely managed by the information systems themselves, but by the overall environment.

1.1.3 What to do

To behave well in nearly every situation the software has to cope not only with the normal situations but also with a number of unexpected situations. Thus the motto should be: *Expect the Unexpected!* Peter G. Neumann also gives this advice. In his book [Neu95, chapter 9] about computer related risks he concludes from an analysis of a number of problem cases:

<p>Expect the Unexpected!</p> <p><i>What we anticipate seldom occurs; what we least expected generally happens.</i></p> <p style="text-align: right;">Benjamin Disraeli</p> <p>One of the most difficult problems in designing, developing, and using complex hardware and software systems involves the detection of and reaction to unusual events, particularly in situations that were not understood completely by the system developers. There have been many surprises. Furthermore, many of the problems have arisen as a combination of different events whose confluence was unanticipated.</p> <p>Prototypical programming problems seem to be changing. Once upon a time, a missing bounds check enabled reading off the end of an array into the password file that followed it in memory. Virtual memory, better compilers, and strict stack disciplines have more or less resolved that problem; however, similar problems continue to reappear in different guises. Better operating systems, specification languages, programming languages, compilers, object orientation, formal methods, and analysis tools have helped to reduce certain problems considerably. However, some of the difficulties are getting more subtle – especially in large systems with critical requirements. Furthermore, program developers seem to stumble onto new ways of failing to foresee all of the lurking difficulties. Here are a few cases of particular interest from this perspective, most of which are by now familiar.</p> <p>...</p> <p>BoNY. The Bank of New York (BoNY) accidental \$32-billion overdraft due to an unchecked program counter overflow, with BoNY having to fork over \$5 million for a day's interest, was certainly a surprise.</p> <p>...</p> <p>Reinsurance loop. A three-step reinsurance cycle was reported where one firm reinsured with a second, which reinsured with a third, which unknowingly reinsured the first, which was thus reinsuring itself and paying commissions for accepting its own risk. The computer program checked only shorter cycles ...</p>
--

During normal operation a program is in a „good state“, but normally the set of good states is only a subset of the set of all technically possible states. So there will be many „bad states“ and we want to avoid the program's running into such a state. Following the idea of „Design by Contract“ [Mey88] to be in a bad state usually means that the program cannot fulfill its contract (either the precondition was not fulfilled or the specified post-condition does not hold otherwise).



desirable	acceptable (normal) behaviour ⇨ „good“ state	bingo! ⇨ nearly impossible state
undesirable	unacceptable (error) behaviour ⇨ „bad“ (erroneous) state ⇨ organized panic	uncontrolled behaviour ⇨ disastrous state

Table 1: Classification of system behaviour and state [Den91]

Table 1 summarizes the classification of possible system behaviour and state. For every application we can distinguish between desirable versus undesirable behaviour on the one side and anticipated versus unanticipated behaviour on the other side. An unanticipated and desirable behaviour is rather unusual but it may happen e.g. in case of an incorrect specification and correct implementation.

With error handling we want to reach the following goals:

- We want to be prepared for the program changing into a bad state. In most cases there will be no return from the bad state to a good state and the program has to be terminated in a controlled way. Of course, whenever possible the program tries to get out of trouble by itself, but normally it would be very difficult to recover from a serious error situation. During specification the boundary between the normal, abnormal and erroneous behaviour is drawn and this boundary remains visible down to the code.
- We want to minimize the set of disastrous states by foreseeing these situations and thus converting them to expected situations.

An error handling concept defines which errors must be distinguished and how the system reacts to these errors. Such a concept can be very simple (for instance, every error results in a user message and the system terminates) or complicated.

Each project needs a precise definition of its error handling concept. It must be intuitive and known to the whole team. Otherwise the conceptual integrity of the system will be violated.

Nevertheless, it is one of the hot topics in nearly every project whether a particular situation is an error or not and how to handle it. Especially as unforeseen cases and effects arise during implementation. It's better to invest enough time in careful specifications of non-standard cases than to spend it with lengthy and maybe fruitless discussions. One team member should be responsible for the error handling and he should be consulted in ambiguous situations.

To make the software robust against bad states we have to extend the software architecture by the following error handling facilities:

- Error detection (mainly a matter of idioms and conventions)

- Error handling (retrying, organized panic, false alarm)
- Propagation of error information
- Administration and collection of all information which will be important for the user or developer to analyze and resolve an error
- Administration of error messages which will be shown to the user in case of errors (error reporting)

1.1.4 How to do it

That is the central question of this paper. The pattern language provides you with a collection of patterns about error handling practice and experience. Within the patterns a number of examples and sample code illustrate the problems as well as the solutions.

In the next chapter we introduce basic terminology together with some examples. You can skip this chapter if you feel sure about the terms and their correct usage. If you need more background information about the context of error handling and error specification (or if you get more appetite) you should read Appendix Appendix C.

Chapter 1.3 gives you a survey of the pattern language in form of a roadmap. It may show you which patterns you have to „visit“ on your way to more reliable software.

After all these preliminaries we finally start in Chapter 2 with the main part of the paper, the pattern language, using a top-down approach. Chapter 2 begins with a framework pattern that explains the architectural view of error-handling in pattern form. This pattern contains many references to (lower-level) design patterns which are also part of the pattern language. They cover different design issues but still on a level independent from a particular programming language.

The final part of the paper consists of design rules, several appendices and a list of references. The appendices contain a glossary, a checklist (which has to be improved), some theoretical background, and an excerpt of a report on the Ariane 5 failure. You can read the appendices independently from the rest of the paper.

1.2 Terminology

So far we have already used a lot of specific terms, like *abnormal situation*, *exception*, *exception handling*, *error* and *fault-tolerant*, but we did not explain, what we really mean with these terms. Furthermore there exist no common definitions, which often results in misunderstandings. So let's start with some definitions (corresponding to [Lap92]), to get a consistent understanding of the basic terms. The definitions are accompanied by a number of motivating examples.

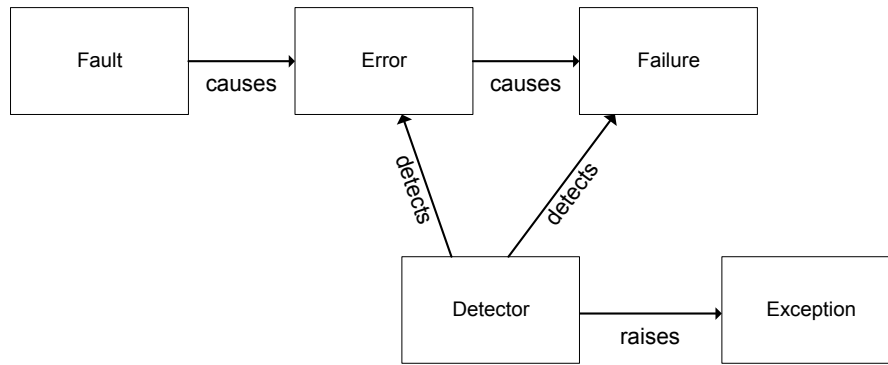


Figure 2: Cause relationship

1.2.1 Fault

A *fault* is the origin of any misbehaviour. It is the adjudged or hypothesized cause of an *error*. One can distinguish between design faults (software bugs), hardware faults, lower level service faults, and specification faults.

Example: The fault which causes the Ariane 5 failure was a specification fault. For Ariane 5 software from Ariane 4 was reused, but this software does not match the requirements for Ariane 5. The validation of the specification against the requirements and the testing of the final software were not sufficient to detect the fault. □

1.2.2 Error

This is the part of a system state that is liable to lead to a *failure*. With respect to a *fault* it is a manifestation (or in object-oriented terms an instance) of a *fault* in a system.

Example: In the Ariane 5 failure the error was a horizontal bias value computed by an alignment function. This value was out of the expected range which lead to a failure in the conversion of that number. This error could have been easily detected by a range check for the computed value. In case of an invalid value an error handler could recover from this error by using the last valid value computed by the function or a predefined constant value. Note that this recovery action does not remove the original problem which causes the error, but it makes the software more fault-tolerant. □

Note that user errors are not part of the above error-definition. A *user error* is a mistake made by a user when operating a software system. The system is able to react to these mistakes, because it is designed to expect such situations and it is a part of the required functionality. The handling of those situations, which of course can be abnormal, should be treated in the user interface component of the system. In contrast to an error, a user error (hopefully) can not result in a system crash, but like a system error, a user error normally can result in an exception raised by a system component.

Example: There is a daily limit for cash from an automatic teller machine (ATM). If a user exceeds this amount, a message which informs the user about this limit is shown on the display and the action is refused. This is an example of an incorrect usage. The validation of this

constraint must be part of the acceptable system behaviour, although this behaviour is not the standard case. Afterwards, the user can continue with other actions. This constraint can also be formulated as a precondition for the method which implements the particular transaction.

1.2.3 Failure

A failure is a deviation of the delivered service from compliance with the specification. Whereas an error characterizes a particular state of a system, a failure is a particular event namely the transition from correct service delivery to incorrect service. Meyer gives the following similar definition: „A *failure* is the inability of a software unit to satisfy its purpose (denial of a service)“.

Example: Continuing the Ariane 5 example the alignment function failed as a result of an unprotected number conversion (an exception was raised indicating an operand error).

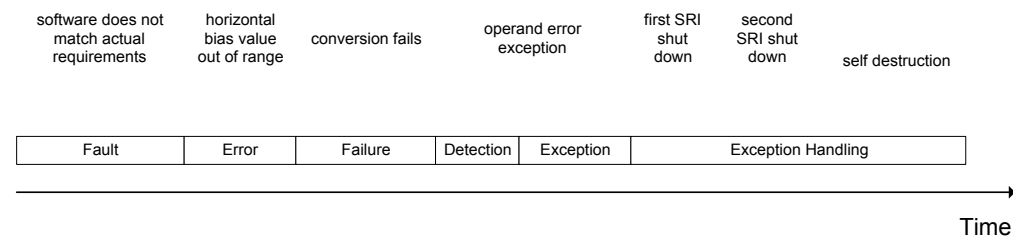


Figure 3: Event chain of the Ariane 5 failure

But there was a specified exception handling mechanism (a failfast approach) which stated that any kind of failure (exception) should be indicated on the databus, the context should be written to an EPROM memory (error log) and finally the processor of the Inertial Reference System (SRI) should be shut down. Thus the unexpected exception does not cause uncontrolled behaviour but the whole SRI fails. The On-Board Computer was able to detect a failure of a SRI and switched to a second SRI. Unfortunately, the second SRI was identical to the first one and therefore failed too. □

1.2.4 Detector

Before software can react to any error it has to detect one first. If we look at the error handling as a separate software system, errors and failures of an application are the phenomena the error handling system observes. Thus a detector is the interface between the error and failures happening outside and their internal handling. The number, place and kind of detectors have great impact on the quality of an error handling system.

Example: Interface Check

An invalid parameter value indicates that a client does not obey the specified contract for that service. This is often the result of a serious programming fault which is handled by termination of the application.

Consider a method `CheckRange` of a class `NormalRange` in a laboratory management system which checks a test result against a predefined normal range for this test. This method gets the result value as input value and computes a classification as a result.

```
ResultClassification CheckRange(Concentration ResultValue)
```

The concentration of a substance is measured in a certain unit – a natural number defined by the datatype `Concentration`. In the beginning of the method body, we check the validity of the parameter `ResultValue` (e.g. by using a method `IsValid` of the datatype `Concentration`) and possibly raise an exception `InvalidParameter`.

Like the check for unexpected input values we can also check for unexpected output values. We can extend the method by an additional check for the `ResultClassification` before a return statement terminates the method call. □

Example: Constraint Check

In the laboratory management system there is the following constraint:

A sample from a patient can only be attached to an order which is also related to this patient.

This constraint can also be formulated as an invariant for the classes which contain the corresponding data of samples, orders, and patients. In the code this invariant can be checked as postconditions of the methods manipulating the data. A violation of this constraint results from an inconsistent database (possibly a bug in the software, a communication failure or a hardware failure). □

1.2.5 Exception

Generally, any occurrence of an abnormal condition that causes an interruption in normal control flow is called an *exception*. It is said that an *exception is raised (thrown)* when such a condition is signaled by a software unit. In response to an exception, the control is immediately given to a designated handler for the exception, which reacts to that situation (*exception handler*). The handler can try to recover from that exception in order to continue at a predefined location or it cleans up the environment and further escalates the exception. To raise an exception, a program first has to detect an error or failure (see Figure 2)!

To deal with all exceptional situations is a thorny problem and one of the main sources of increasing complexity of software. Without a systematic approach and careful design the extendibility, understandability and maintainability will be decreased. Note that the exception mechanisms offered in modern languages are no solution to these problems, they „only“ support the implementation of exceptions by means of better structuring of the code and special control-flow in case of exceptions. You cannot write good software without a good specification (see Appendix Appendix C).

This is what good software design is about: to find simple and elegant solutions which avoid or minimize complex, exceptional cases. It needs experience and invention and cannot be automated.

Note, exceptions are not only a mechanism which can be used to handle errors and program failures. The definition only says that exceptions are used for handling abnormal situations and in this respect they are a general control-flow mechanism, which can also be used to signal other kinds of information (see [Goo75]).

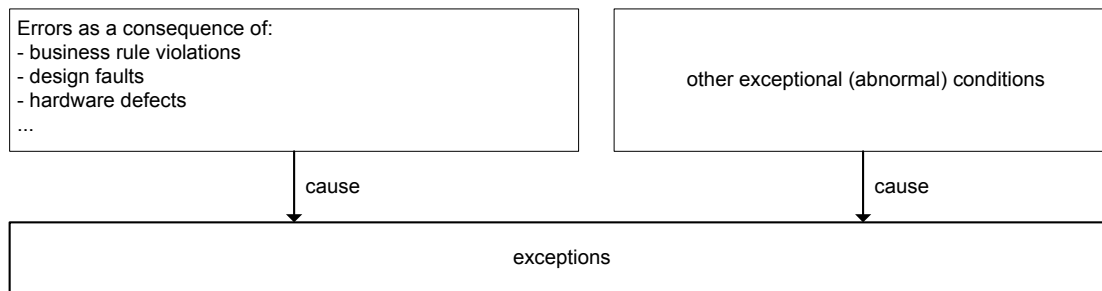


Figure 4: exceptions as a general mechanism

1.2.6 How are these terms related to each other?

A fault in a software system can cause one or more errors. The latency time which is the interval between the existence of the fault and the occurrence of an error can be very high, which complicates the backward analysis of an error. The situation is even more troublesome: on the one hand an error can be caused by more than one fault (see Figure 2), on the other hand an error can lead to one or more failures. Again, a failure can be caused by more than one error.

The chain of errors and failures caused by a single fault is also called error propagation. For an effective error handling we must detect errors or failures as early as possible. If an error or failure is detected it will normally result in an exception which is propagated to a suitable exception handler. The relations are summarized in Figure 2.

Example 1: Retrieve Operation

Imagine a method `RetrieveObjectWithKey(aKey)` which reads an object from a database. As a precondition of this method we can specify that the object defined by `aKey` must be contained in the database. If this is not the case the method raises an exception (`UnknownKey`). For the test whether the element exists another method is defined: `ExistObjectWithKey(aKey)`, which returns a `Boolean` and can be used to check the database for the existence of the particular object. Now the caller can decide whether in his context he has to distinguish and handle both cases properly or he may know that the object must exist, so that an exception would be correct and indicates an error. Sometimes these exceptions are also called failure exceptions, because it is probably a failure. But whether it is really a failure or not depends on the context in which the client uses a service. Note that this is different to a design fault (see next example).

Meyer [Mey88, chapter 9.3] gives the following advice: „whenever applicable, methods for **engineering out** failures are preferable to methods for dealing with failures once they have occurred.“

The code for the client of this service may look like¹:

```
void DoSomethingInteresting(aPatientID)
{
    try {
        if (ExistObjectWithKey(aPatientID) {
            ...
            Patient = RetrieveObjectWithKey(aPatientID);
            ...
        }
        else {
            // create new object with aPatientID ...
        }
    }
    catch(...) {
        // UnknownKey exception is handled as a failure
    }
}
```

You can often find other specifications of these methods. A common approach is a combination of both methods: the `RetrieveObjectWithKey` method is extended by an output parameter for the object and a return value of type `Boolean` for the existence. This is of course not so flexible. Sometimes this solution is preferable because of performance, but often it is not necessary.

Another variation which is very common avoids the `Boolean` return value. A pointer to the retrieved object is returned and when the object could not be found the pointer is `nil`. The caller is responsible for the correct interpretation of the `nil`-pointer. Somehow the `nil`-pointer is misused to signal other information by the same return type. For the readability and reusability of the code the other solutions should be preferred.

With respect to conceptual integrity do not mix these approaches within one application. You have to choose one and this choice must be documented as a design decision. All team members must be aware of this design decision to get a coherent design! □

Example 2: Disk Full

Consider a method which writes data to a file on a disk. This method gets the full pathname as input. When not enough disk space is available the method raises an exception `DiskFull`. This is a method of a lower level service class which is used by a number of different applications. Now we can consider different application scenarios:

- 1) An installation program will check the available disk space before starting the actual installation. If the write method still raises a `DiskFull` exception it must be handled as a serious error. For instance, the computation of the disk space could be incorrect, but the write method would not know. So the `DiskFull` exception would correspond to the specified behaviour and therefore this wouldn't be an error of the write method! But of course there could also be a fault within the write method.

¹ Note, this code wouldn't be acceptable if both method calls access a database. We assume that there would be a cache which avoids this situation here.

- 2) Another application highly depends on the availability of enough space on a hard disk. Therefore the system operates on two hard disks which are managed by a fault tolerant method. This method catches the `DiskFull` exception of the lower level write method and then switches to the second disk. Only if writing to the second disk yields a `DiskFull` exception the method also raises an exception, which then must be handled as a serious error.
- 3) The write method also can be used by an application to store some user data to a floppy disk. Because the situation of *not enough disk space* won't be so serious, the available disk space is not checked before a file is written to the disk. The `DiskFull` exception is caught instead and is handled by an appropriate user message. The user has the choice to insert another disk or to cancel the write operation and to proceed normally. This is example for an exception from a lower level component which does not result in any error or system failure. A variation of this example is that a user forgets to deactivate the write protection of a floppy disk. This can also lead to an exception of the lower level write method, although the system finally proceeds with normal operation. It is an example for a user error handled via exceptions.

□

1.3 Pattern Roadmap

Following a conceptual point of view, we can extract an architectural pattern for error handling which is valid for nearly all implementation languages. We also call this pattern a framework pattern because it describes the problem and its solution on a unified and broad abstraction level.

We can look at this level from two different angles, each of which provides us with a different view on error handling:

1. *Infrastructure view*. It concerns all kind of common services which are necessary for a suitable and effective error handling. Together these services build a separated component within an application.
2. *Installation (or robust component) view*. This view gives answers to the question: how do you convert a component into a robust component by using the error handling infrastructure? Important aspects are detection, handling and propagation of errors.

For the refinement of these views a number of design patterns are given. The infrastructure view is much more independent from the implementation language than the installation view. Therefore, we also present some patterns which are useful for the implementation of the various aspects on the installation side. Generally, we can distinguish between 3GL languages (e.g. C, Cobol, PL/1) and object-oriented languages on the one hand and between compiled languages and interpreted languages (Smalltalk, Java) on the other hand. It is not a big surprise that the application and implementation of the patterns in languages like Java or C++ is much easier and straightforward than the implementation in a language like Cobol. For instance, today's languages offer features like:

- Exceptions

- Powerful debugging (remote debugger)
- Dynamic facilities (meta information, closures)
- Runtime environment (which is able to detect a lot of errors)
- Garbage collection (which prevents another group of common errors)
- Dump of the current stack content

Therefore, these languages are much closer to the conceptual error handling model and provide us with a good, basic infrastructure which eases the development of robust software. In other languages we partially have to build this infrastructure by ourselves. The additional steps (mappings) needed to implement these concepts can often be automated by use of macros and generator tools (an example is extended Cobol, abbr. xCobol [TLR95]). Of course these often extend the existing languages by missing abstractions and constructs which are built-in by the next generation language.

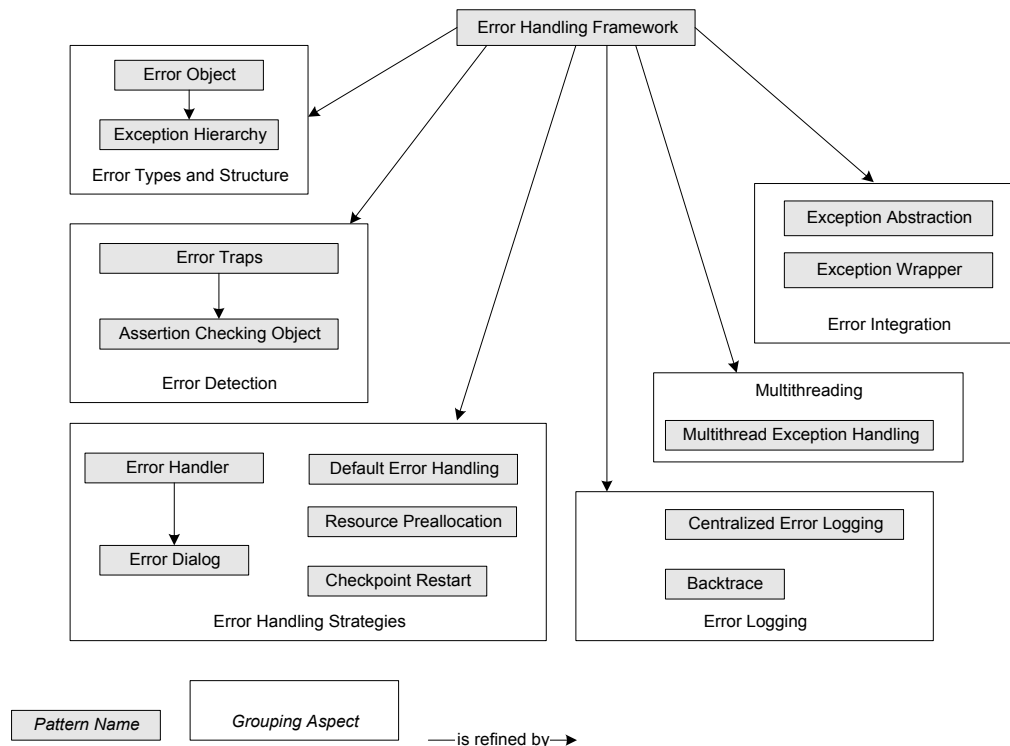


Figure 5: Map of the pattern language

Figure 5 presents a synopsis of the pattern language. The central pattern is the *Error Handling Framework* surrounded by a number of patterns on a lower abstraction level. The error handling problem space is decomposed into different problem domains (Error Detection, Error Logging, etc.) and the lower level patterns are grouped according to these domains. Chapter 2 reflects this structure as every domain becomes a subchapter.

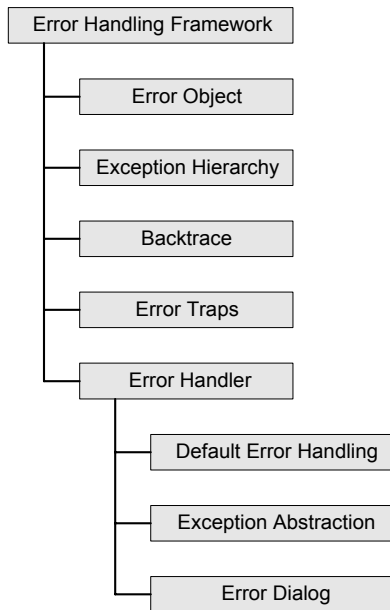


Figure 6: Basic patterns

It is not necessary (and even not advisable) to sequentially read over all patterns at once. Each pattern can be read independently. If you are familiar with error handling and you are just looking for a solution to a particular problem, search through the pattern catalog in Chapter 2 or have a look at the checklist in Appendix Appendix B. We can distinguish between basic patterns, which provide the essence of error handling, and advanced patterns, which deal with special problems. If you want to learn about common error handling read the basic patterns as shown in Figure 6.

1.3.1 Pattern Catalog

To get a better idea of the various patterns and the problems they are related to, we list all pattern names together with page references and questions defining the problems:

Error Object (31)

What characterizes an error? How to structure and administrate error information?

Exception Hierarchy (37)

How do you structure error types? What role does inheritance play in the structuring of errors?

Error Traps (42)

Which indicators are useful to detect erroneous situations and where should the traps be installed in the application code?

Assertion Checking Object (50)

How to implement Error Traps in an object oriented language without using a generative approach?

Backtrace (55)

How do you collect and trace useful information that helps the system developers or the maintenance team analyze the error situation? Especially when we have no or limited access to the stack administered by the system itself.

Centralized Error Logging (61)

How do you organize exception reporting so that you can offer your maintenance personnel good enough information for analyzing the branch offices' problems?

Error Handler (80)

Where and how do you handle errors?

Default Error Handling (65)

How do you ensure that you handle every possible exception correctly (no unhandled exception and limited damage)?

Error Dialog (68)

How to signal errors to an application user?

Resource Preallocation (84)

How to ensure error processing although resources are short?

Checkpoint Restart (76)

How do you avoid a complete rerun of a batch as a result of an error?

Exception Abstraction (90)

How do you generate reasonable error messages without violating abstraction levels?

Exception Wrapper (93)

How do you integrate a ready-to-use library into your exception handling system?

Multithread Exception Handling (98)

How do you schedule exceptions in a multithread environment?

1.4 Notational Conventions

Concerning notation the patterns contain OMT-diagrams [RBP+91, GOF95] to describe static object structures. For interaction scenarios we use a notation based on the **Object Message Sequencing Charts (OMSC)** of [BMR+96] with minor extensions for the context of error-handling:

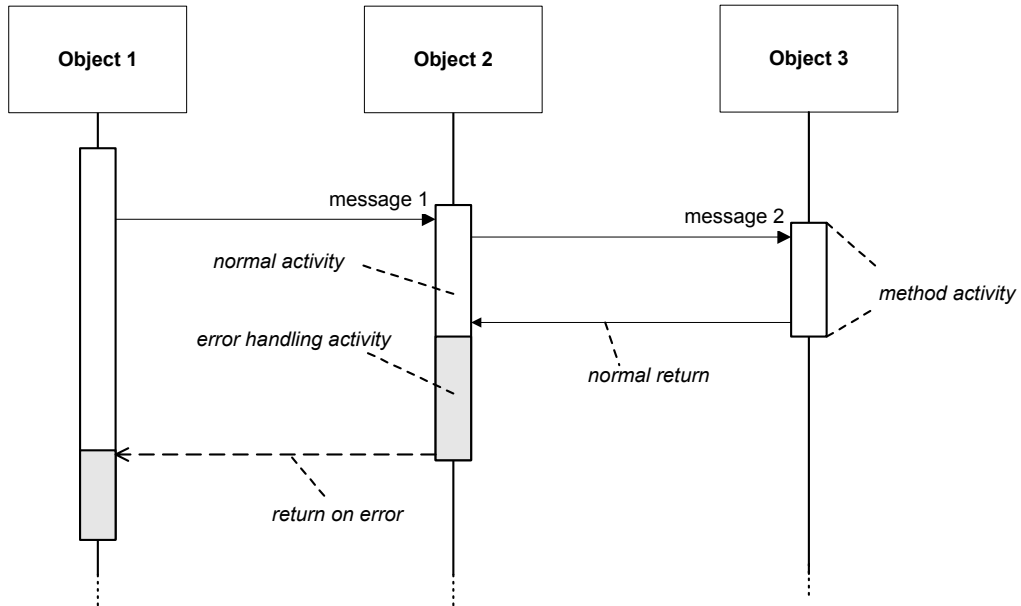
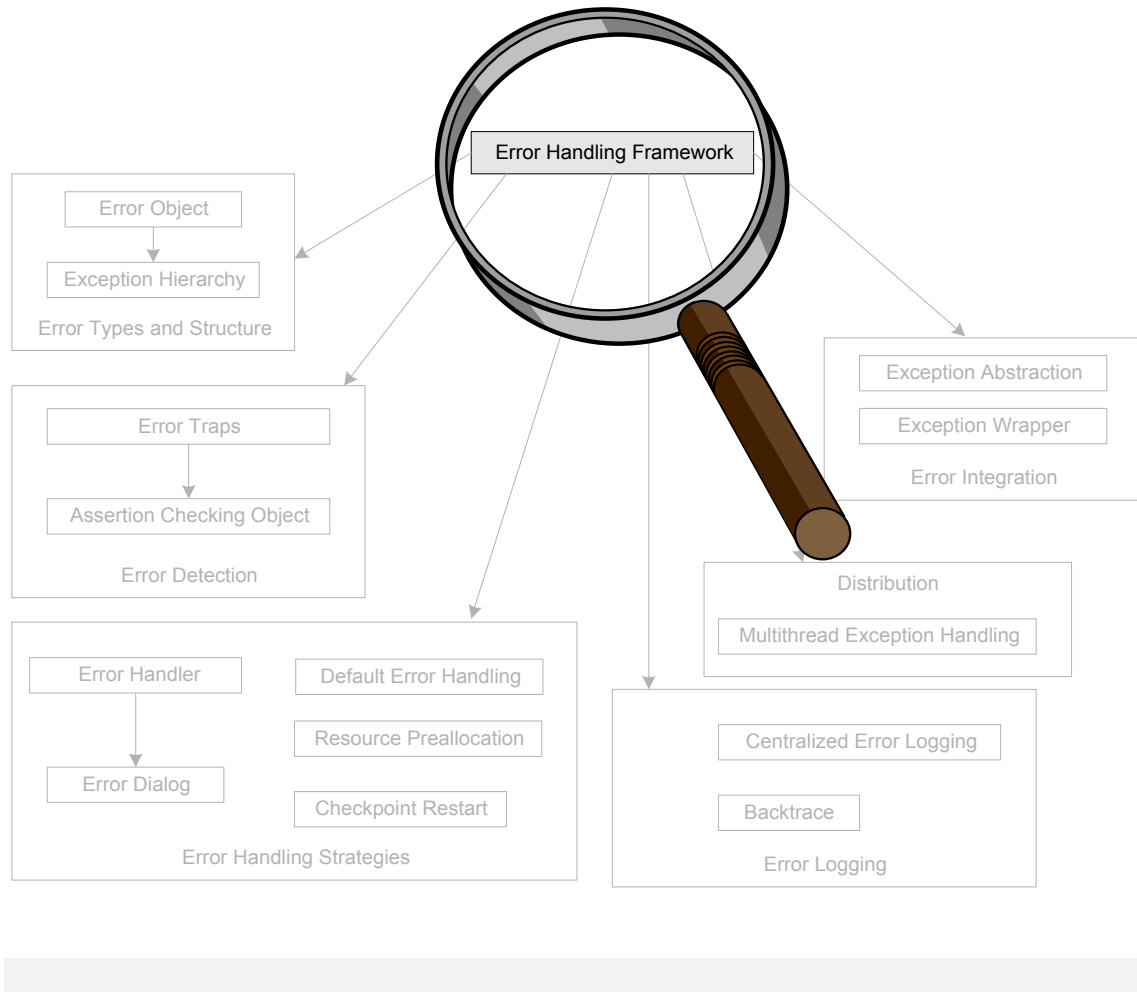


Figure 7: Extended OMSC notation

2

Pattern Language

2.1 Architecture



Error Handling Framework

Abstract

The Error Handling Framework defines a basic infrastructure and mechanisms for building more reliable and fault-tolerant information systems. It integrates error handling facilities into the overall system architecture of an information system.

Example

In the laboratory management system the following error may occur: the analysis instrument reads a sample's bar-code to get an identifier for that sample. But instead of the correct sample identifier a wrong value is computed. This value is used by the instrument to ask the laboratory management system for the test requests ordered for this sample.

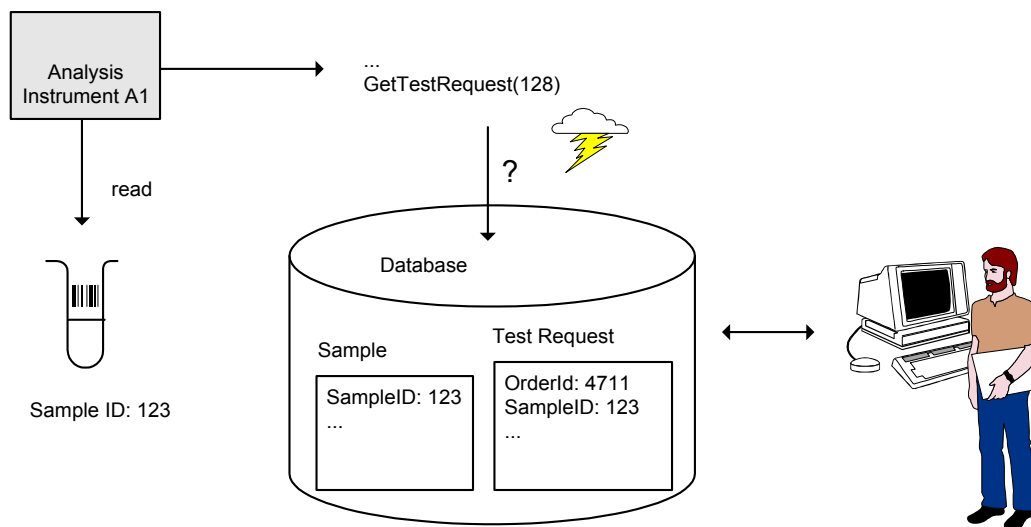


Figure 8: Error Scenario

Because of the wrong identifier, no sample and also no related test requests exist and therefore the access fails (see Figure 8).

But of course we wouldn't accept the system to crash due to this error. We expect the system to be able to detect this error and finally informs us properly about this erroneous situation through a user message.

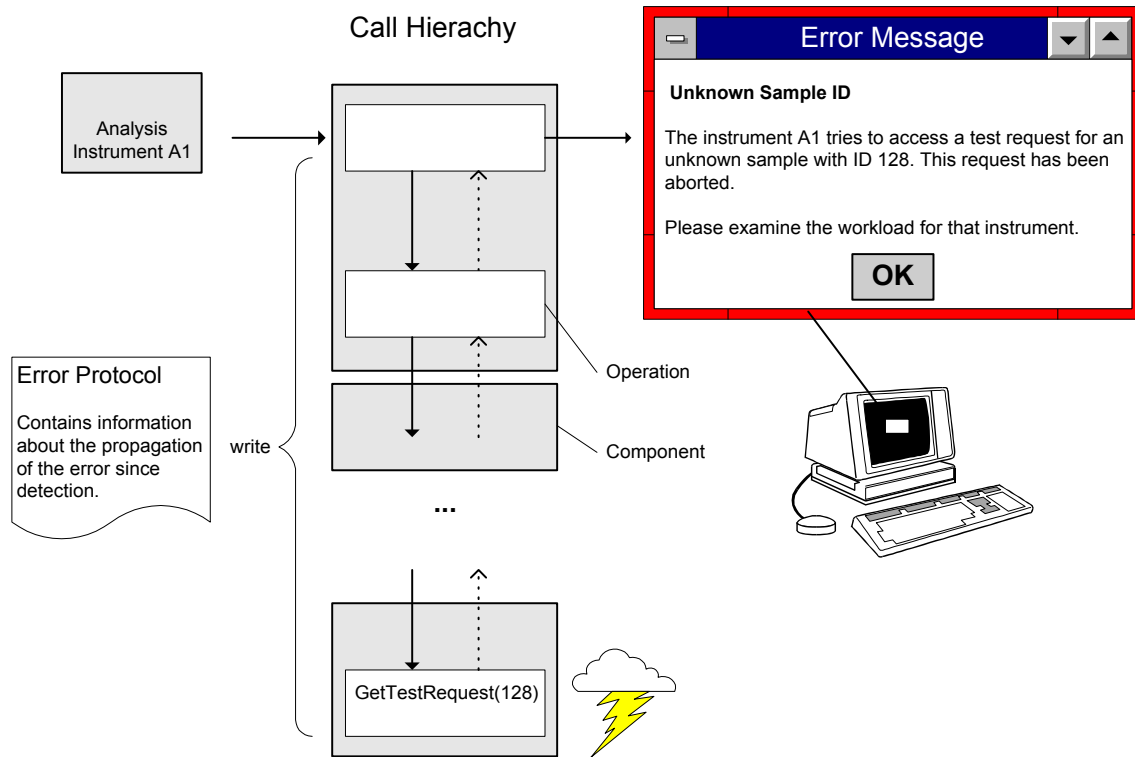


Figure 9: Desired system reaction

Although the message advises the user about possible solutions to the problem, it is not sufficient for the technical backward analysis of the error by a system developer or administrator (Figure 9). Thus we also need a more detailed description of the program's state and behaviour since detection of the error (called error protocol or error log).

Context

No software system can be assumed to behave totally correct. Even careful testing and the use of formal methods does not guarantee error-free software, hence we cannot rely on a „perfect world“ assumption.

Because we cannot prevent errors we have to live with them. The software must be able to detect errors and to defeat them with appropriate recovery techniques and mechanisms in order to restore normal operation.

This pattern applies nearly to any kind of large information system, where reliability and fault-tolerance are important issues (this may not be the case for a prototype).

Be aware of the fact that this architectural pattern outlines a general concept and infrastructure for error handling. If you use this pattern, it does not guarantee that your program will behave well in nearly every situation. It depends heavily on your specification and on the actual implementation of your error handling code. How you should react to an error in a concrete situation is often the business of your application and cannot be generalized within this architectural pattern.

Problem

How do you design a reliable and fault tolerant system which

- keeps track of error situations in a detailed manner to support the development and the maintenance of the software and
- is able to inform the user about errors by suitable error messages?

How do you integrate the necessary error handling facilities into the system architecture?

Forces

- *User interaction*: Even in erroneous situations the system should behave in a controlled way and the user should be informed appropriately about the system's state. You have to take care of the interaction between the error handling in the application code and the user interface to avoid cyclic dependencies.
- *Robustness*: The error handling should be simple. All additional code for handling error situations makes the software more complex, which itself increases the probability of errors. Thus the error handling code should provide some basic mechanism for handling internal errors. However, for the error handling code it is even more important to be correct and to avoid any nested error situations.
- *Separation of error handling code*: Without any separation the normal code will be cluttered by a lot of error handling code. This makes code less readable, error prone and more difficult to maintain.
- *Specific error handling versus complexity*: On the one hand we have to classify errors more precisely to handle them effectively and to take measures tailored to specific errors. On the other hand the error handling becomes more complex, which also influences *Decoupling* and *Robustness*.
- *Detailed error information versus complexity*: Whenever the system terminates due to an error we need suitable information to analyze the error and to manually restore the system. Otherwise, it is not feasible to investigate the original fault that causes the error. However, the more detailed the error information, the more error handling code we have to write.
- *Performance*: We do not want to pay very much for error handling during normal operation.
- *Reusability*: The services of the error handling component should be designed for reuse because it is a basic component useful for a number of applications.

Solution

Make error handling a „first class“ component of your system architecture: build a reliable and reusable component („Reliable Computing Base“²) for common error handling services and secondly enrich all other system components by error handling code using these common services.

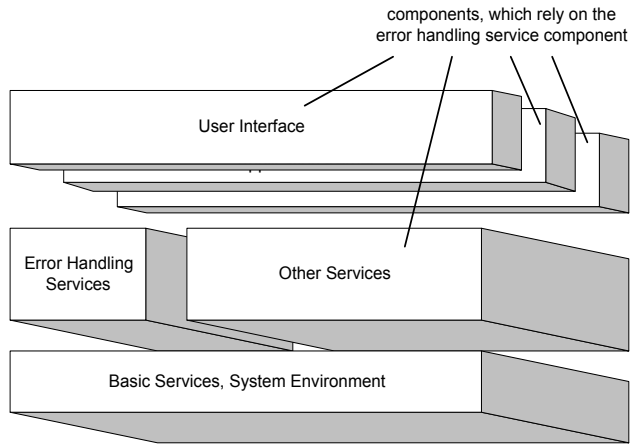


Figure 10: Integration of an error handling component—separated from the normal communication bus. The error handlers of the connected components are the fire-detectors within the system and the error bus is the fire hotline.

Structure

As the client classes are structured by layers also the error handling component consists of different layers. It contains classes responsible for user interaction, technical classes (e.g. `Error`, `ErrorProtocol`), classes for data access and finally there are some base classes. The component itself is based on lower level services offered by the operating system itself or class libraries and frameworks on top of it. However, these low-level classes do not rely on the error handling services and may also be used by the other application components. The error handling takes special care of failures signaled by lower-level services in contrast to the other application components.

The use of the error handling component by the clients must be consistent with the layering.

² All the error handling services build a *Reliable Computing Base* comparable to the notion of a *Trusted Computing Base* (TCB), a term used in the field of computer security.

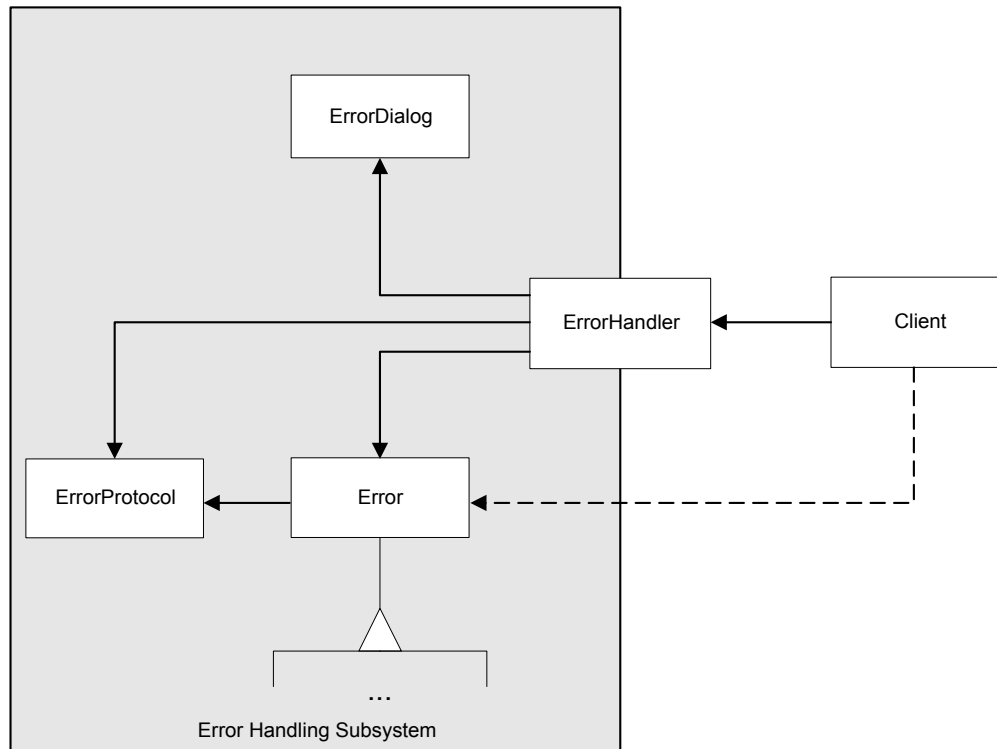


Figure 11: Structure of the error handling participants

Participants

- **Error**
 - An Error Object encapsulates all information about an error and its context.
 - As a base class you can derive more specialized error classes from it to get an Exception Hierarchy.
 - Clients create and use error objects like other datatypes.
 - An error objects knows how to write itself to the `ErrorProtocol`.
- **ErrorProtocol**
 - Is a Singleton [GOF95] responsible for writing data to a log file.
 - Records detailed error information including a dynamic call chain, which reflects the control-flow up to the point where the error was detected.
 - It is adjustable by a number of configuration parameters which are used by the methods of this class to control the output (e.g. whether information should be flushed immediately or buffered).
- **ErrorHandler**

- This class is a Singleton [GOF95] as well as a Facade [GOF95] for the error handling subsystem.
- The Error Handler encapsulates error processing and helps to enforce consistent error handling strategies.
- A `Client` uses `ErrorHandler` to process any errors.
- The `ErrorHandler` uses `ErrorDialog` to display error messages.
- The `ErrorHandler` uses `ErrorProtocol` to generate an error protocol.
- **ErrorDialog**
 - The Error Dialog is responsible for displaying error messages on the screen.
 - Generally this dialog will be a modal dialog.
 - The `ErrorDialog` uses lower level services (e.g. GUI-framework).
- **Client**
 - Once an error is detected the client creates an `Error` object and supplies necessary context data.
 - A client's method signals a failure to its caller and passes an `Error` object to him.
 - If the client is a control object within the user interface he may use the `ErrorHandler` to display a message for the error which was signaled to him by another method.

Example Resolved

The following diagram illustrates a typical interaction scenario. There are two client objects for the error handling services: first an interface object for the analysis instrument A1 and second the workload for this instrument.

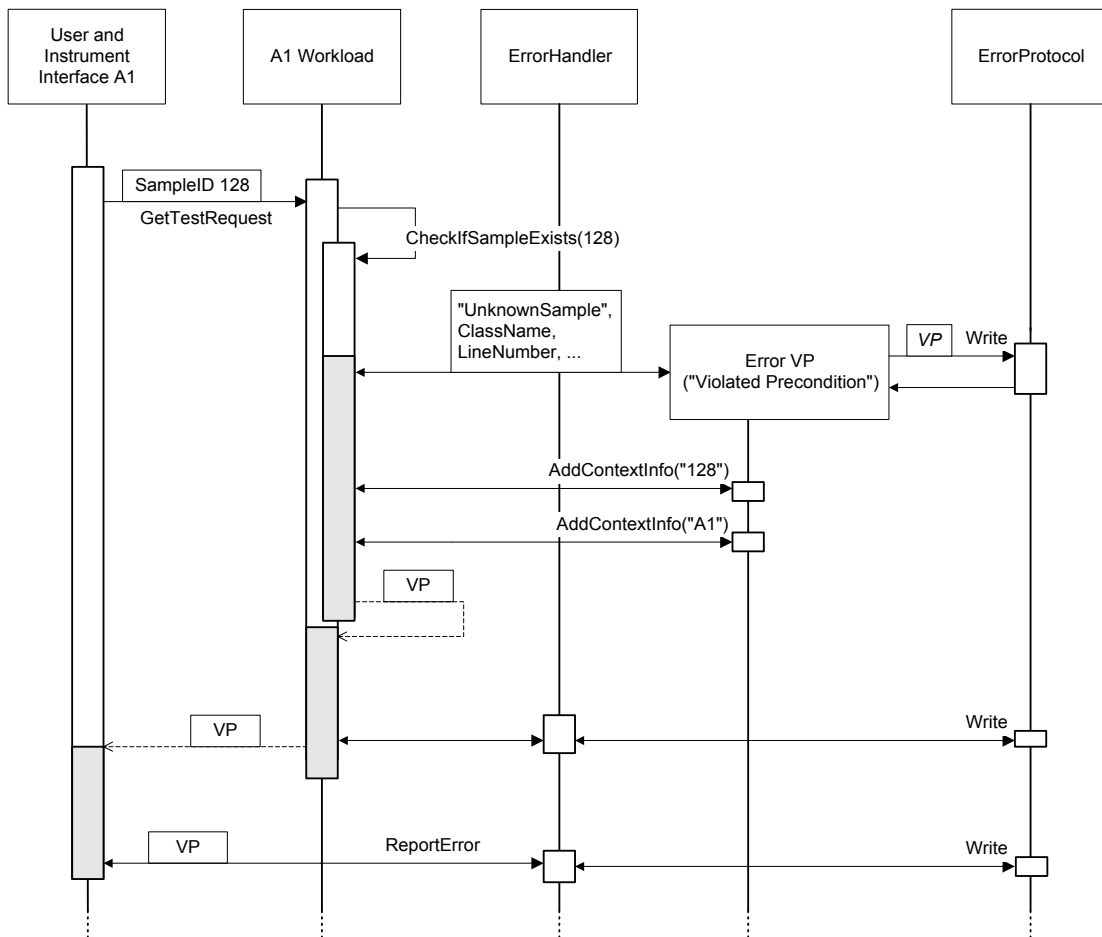


Figure 12: Collaboration of the application classes and the error handling subsystem.

The scenario (simplified) comprises several steps:

- The interface object asks the Workload for a test request for the current sample with ID 128.
- While processing the message `GetTestRequest(128)` the Workload checks the existence of the sample 128. The specification of `GetTestRequest` contains a precondition which says that the result is only defined if a sample exists for the given ID.
- Because the check failed an `Error` object VP is created. The constructor is supplied with the necessary data for its parameters.
- The constructor writes the error object to the log.
- The VP object is signaled to the interface object as a result of its request.
- To trace the flow of information additional data is written to the error log via the `ErrorHandler`.
- Finally, the `ErrorHandler` reports the error to the system user.

Consequences

- *Controlled Panic*: The pattern enables to shut down a system gracefully even in a serious error situation. Error messages inform the system user and an error log records detailed information for off-line analysis.
- *High Reliability*: To prevent any errors to occur within the error handling services, which may also result in cyclic error loops, the pattern tries to make the services extremely fail-safe: it does not allow the error handling services to use other common services of the application which again depend on the error handling services. As many errors as possible caused by lower level services are resolved. If this is not possible, the program terminates (e.g. with some primitive error message).
- *Supports encapsulation and classification of errors*: The error class encapsulates all information about an error. This class can be used as a base class for a more fine-grained classification of errors by means of inheritance.
- *Performance*: We have to distinguish between normal and exceptional behaviour. Because errors are really exceptional situations, performance is no longer so important once an error has been detected. Issues like reliability and correctness are more important. Thus the operations necessary for error logging, handling, and reporting don't need any performance tuning. But errors have to be detected before starting any error handling. The client's code is enriched by a number of checks, which results in lower performance during normal operation. You should be aware of these consequences for the performance of your system. Of course, the costs also depend on the programming language and how the checks are implemented within that language. You have to balance the performance and reliability requirements (depending on the criticality of the application).
- *Increased Complexity*: The pattern tries to hide most of the complexity within the error handling services, so the client classes only contain a minimum of error handling code. Nevertheless, code for error detection and calls to the error handling services must be added to each method (method „instrumentation“).
- *Implementation overhead*: Keeping the error handling component as simple as possible can cause some implementation overhead. For example, we need some message handling services to signal error messages to the system users. Also, outside the error handling component we need message handling to inform the user of certain situations (for instance, help messages). As we do not include the whole message handling in the error handling component, there is some overhead in implementing two separate message handling services. A list of desirable features which could make the error handling more luxurious often also exists, e.g. the error log. We also do not include these features in the basic error handling component, but we can build additional components or tools around this basis although this may result in some overhead again.
- *Integration*: Although this pattern describes sd&m standard design it will not be compatible with a lot of software components which we have to use and integrate within the system (e.g. libraries, other applications, frameworks). Thus we are faced with the problem of how to connect these components to our error handling infrastructure.

- *Parallel Processing, Distribution*: The pattern is closely related to the idea of method (or procedure) calls and the unwinding of the call-path with a backtracing of possibly all important information when an error occurs. Thus the pattern depends on a sequential procedure-call model. In other environments you may have to think about other types of exception handling. Be especially careful if your system is highly distributed and the processing is not sequential (the call-chain crosses thread, process or workstation boundaries). In this case we need enhanced error handling facilities or even other concepts (e.g. an active observer component which permanently controls the state of other system components).

Implementation

There are some implementation issues to consider:

- *Configuration*. To be flexible and reusable the error handling component has to be dynamically configurable by some parameters. The configuration data can be made persistent via configuration files or a database. Possible configuration parameters are:
 - *Checking Intensity*. It is likely that the need for error checking changes during the lifetime of a software product. During development and testing the probability of errors is much higher than in later phases. Although the customer might require to be able to switch off or on certain tests without any recompilation.
 - *Error Log*. For the error protocol a lot of adjustable control parameters are useful (e.g. protocol on /off, name and location of the log file, maximal size, types of errors and components for which to write a log, text template for the log file)
- *Error detection*. Before we can think about how to handle certain errors we have to find them (Error Traps). Therefore, we enrich our classes by checks for various properties which must be satisfied. Different solutions are possible:
 1. The programming language already offers special check instructions. For example, the programming language Eiffel [Mey88] offers a *check* instruction to formulate common assertions. Preconditions are expressed separately by a *requires* instruction. Whether the asserted properties are actually monitored at run-time can be controlled by compile options. It is possible to specify different check levels on the granularity of single classes. Depending on the specified options, the Eiffel compiler generates the monitoring code.
 2. We explicitly implement some check and monitoring functionality (e.g. like that in Eiffel) by using the features of the programming language (Assertion Checking Object).
 3. We „extend“ a language by macro commands which are converted to the base language by a macro processor.

A combination of these variants can also be useful.

- *Error handlers*. The code within the clients becomes more complex because of the error handlers (Error Handler). You have to look for the readability of the code. This can be

achieved by clearly separating the normal code from that for the exceptional cases (don't mix a cocktail). Also see Chapter 3 for additional information.

- *Flexible error logging*: For the output of useful information about an error we extend nearly every class by a special method which converts the content of an instance of that class to a string. This string can then be written to a log file. Because every error is also a class (Error Object) containing this method and encapsulating all the necessary information the conversion to a string is used to write this information to the error log. Thus an exception object is written to the log file like a return value to a trace file. This allows a very flexible logging of error information and is also very similar to the more general tracing.
- *Nested Error Situations*. We cannot exclude that an error occurs while handling another error. Even if this situation of nested errors seems to be very unlikely, we should expect such a situation and provide a (last) emergency exit which prevents a system crash. The main problem here are not design flaws within the error handling component, because it is a small and well-tested component, but errors within the basic components we must rely on. For instance, Resource Preallocation offers a solution for fault-tolerance against „out of memory“ situations. If fault-tolerance is impossible, we terminate the application and try to signal a last message to the user (e.g. by a standard output device).
- *Integration*. If a system reuses or composes existing programs, frameworks, and libraries, its error handling concept likely differs from that outlined by this pattern. Therefore, you have to think about how to bridge and integrate the different approaches which may result in a mapping between different error classifications. For example, if a library uses return codes they have to be converted to corresponding exception objects. The Exception Wrapper pattern gives more advice.
- *Exception Abstraction*. The propagation of errors should not violate your abstraction layers, so you also have to abstract from errors on the interface of system components. For example if a component hides the information that it uses a file to store some data and an error occurs so that the file could not be written, the error information given to the client of this component should not mention any facts about the file. It would possibly only reveal that the data could not be stored, so that the abstraction is not bypassed.
- *Resource Management*: Resource management is an important topic for error handling and especially for recovery (see Chapter 3). If the programming language does not support garbage collection, resource management can be a time-consuming activity (e.g. in C++). To prevent fatal *Out Of Memory errors* (especially within the error handling component) Memory Preallocation could be helpful.
- *Multithreaded environment*: Problems with error handling while using threads can arise, because the error handling normally is processed on the local stack of a thread. So we must manage the propagation of errors between threads, synchronize the access to the global exception handler, and produce a consistent and complete exception log. If a multithreaded application must be terminated in case of an error it must be ensured that all threads will clean up and shut down gracefully. The detailed solution to these problems depends on the programming language and its support for multithreaded application with respect to error handling. However, the error-handler must distinguish which thread has raised an error exception. See Multithread Exception Handling for more information.

- *Distributed environment:* In a distributed environment [Central Error Logging](#) can be used to support maintenance. It offers easy access to error logs while the software is running on PCs in different sites of a big company.
- *Log Inspection Tools:* Whenever a serious error occurs the system produces an error protocol. With these protocols at hand developers and system administrators must reconstruct the situation which causes the error. Error protocols are simple ASCII files written by a program in a critical emergency state. How can we support the post-mortem analysis with these error protocols so that a bug can be found and fixed quickly? Imagine a Java application for inspecting error protocols. This may be useful for system administrators to read error protocols at their local sites whereas the error protocols are written on the server ([Central Error Logging](#)). The error protocols may be written in HTML format just by the error handling component or there are converted to HTML by the tool. Once an error protocol is written, a mail might be generated automatically to inform the system administrator. It is not necessary to attach the error protocol to this mail even if the inspection be done by developers located elsewhere; they just start their browser and examine the error log. Every method name might be a hyperlink which is resolved by the tool and enables to view the corresponding source code (e.g. in cooperation with WIORA).

An example of a classical tool which supports post-mortem analysis is the dbx debug tool in a unix environment. By calling dbx with a core file as an argument it is possible to examine a program state when the core dump was produced.

Another functionality of an inspection tool could be to match a new error log to a set of previous error protocols and analysis results which are stored in an error database. This might help recognize similar errors and to fix a bug more quickly. Of course, the new error log is added to the database by the tool. The error database and the tool can be used during testing as well as during maintenance to keep track of errors and thus to improve quality management. Generally such a tool would be a great enhancement for the software development environment.

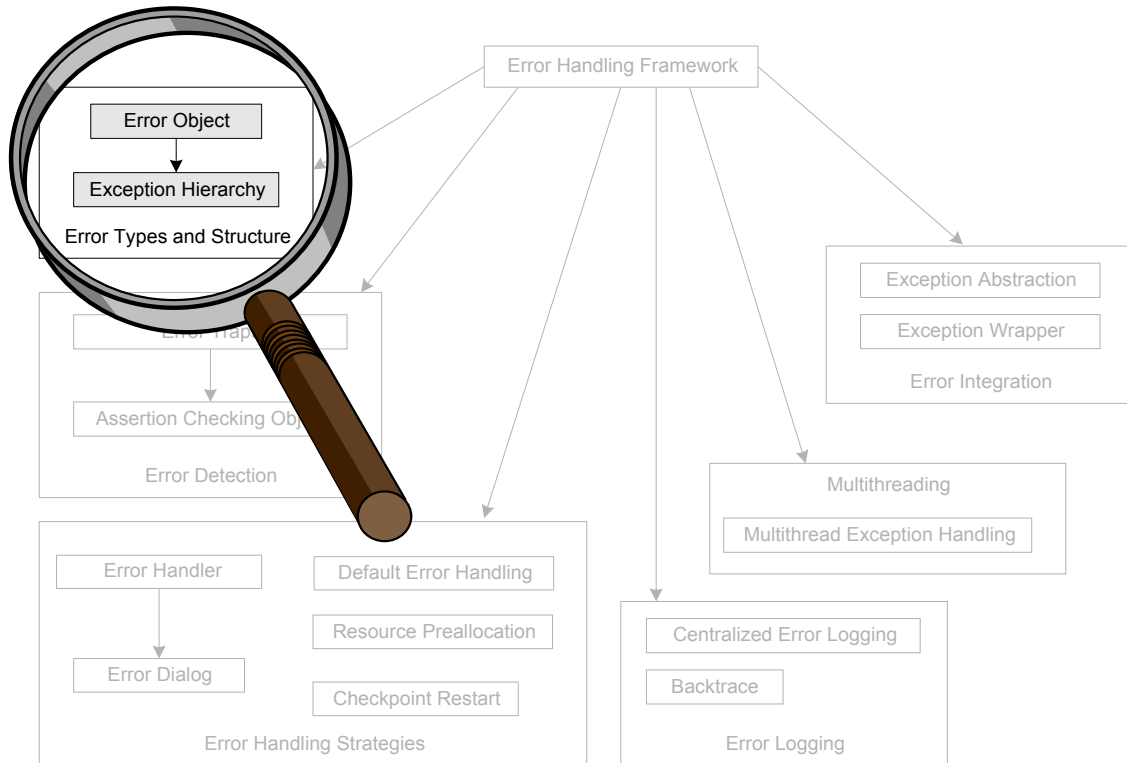
Known Uses

This pattern describes „the standard“ pattern for error handling at sd&m, which has been sketched earlier by Denert [Den91, pp. 297-311]. It is a well-trying solution due to a rich number of sd&m projects which use this architecture.

Related Patterns

The component is a separated „column“ standing aside the other components of an information system. The interior structure of the error handling component is built by a variety of patterns and not all of them are specific to error handling but can also be used elsewhere. Especially administration of a log file, message handling and configuration are common topics. Others (e.g. [Exception Wrapper](#), [Exception Abstraction](#)) are concerned with exception handling generally.

2.2 Error Types and Structure



Error Object

Abstract

The pattern argues about introducing a separate class to model errors. An error class can encapsulate all kind of relevant error information.

Context

Object-oriented design of an error handling infrastructure. As errors play a central role within the error handling concept, we are considering how to model them in our design.

Problem

What characterizes an error? How to structure and administrate error information?

Forces

- Two different groups of people are concerned with errors: system users interested in the functionality of the system and technically skilled people responsible for e.g. operating and maintenance. Both groups have to be informed about an error. On the one hand we need a representation for errors on the user interface and on the other hand we need technical information which is written to the error log.
- Error information should be easily signaled to a caller of a method.
- We want to avoid the use of global data, but nearly every method can produce an error and therefore must store error information.
- It must be possible to distinguish different kind of errors.

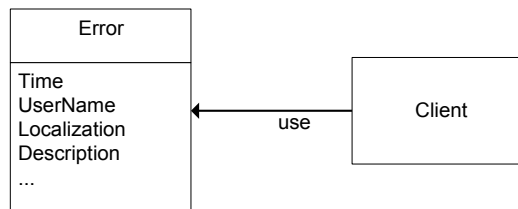
Solution

The idea is to make errors first-class objects. We look at an error as a datatype like any other datatype of an application. Parameter and return values of methods can be of this error type. Therefore, we create a separate error class, which encapsulates the following data:

1. *Localization information* is needed to determine the exact place of an error detector in the software system, e.g.:
 - line number
 - method name
 - class name
 - file name
 - DLL-name
 - program name
 - host name
 - process id
 - revision

2. A *user name* can be useful as it becomes necessary to ask the user additional questions about the error context.
3. *Detection time* helps to analyze an error, especially if an error is related to timing aspects.
4. An *error description* is necessary to explain the error situation and its context.

Structure



Consequences

- *Maintenance*: All data is encapsulated by the class, which increases maintainability and avoids any global data.
- *Access to error information*: Datatypes can be used by every component. We can create an instance of an error class and this instance can be passed to other objects.
- *Information hiding*: We can specify the interface of the error type and hide the actual implementation (e.g. persistence of the error description text).
- *Too many error classes?*: If every error is also implemented by a class, we get a large number of classes. As a result the resource consumption increases: more code, larger executable, lower performance.
- *User defined error types*: We are able to store a chain of error objects and are not forced to report errors immediately.

Implementation

- *Where to get the information from?* Whether all this information can be used depends on the programming environment, maybe some information will not be accessible. In C++ the line number and the module name will be available by special macros (`__LINE__`, `__FILE__`) which are set by the preprocessor. If a version/revision control system is used also information about the version of a module or class might be available by special variables (e.g. `$VERSION$`). You can use these variables to define constants for the versioning information of a module or class so that it is accessible by an error object. If line numbers can not be used, we can number error detection points and use these sequence numbers for identification.

- *Error types.* It is not always necessary and practical to map an error type to a class type in the implementation language. Alternatively, a general error class can be used for a number of different error types. The error class has an attribute which identifies an error type, e.g. an error number or name. We can also use a template to define a generic error class and concrete error types are defined (e.g. via typedef in C++) by instantiation of the error template.
- *Parameterization of descriptions.* For more flexibility and precision we can parameterize the descriptions texts by context-dependent information. Thus the error text becomes a generalized template and everytime the template is used the parameters must be instantiated by the correct values. It is important how to implement these templates and how to ensure type conformance of the parameters.
- *Changing descriptions.* If we want to change error description texts, it is important to know where the descriptions are stored. If they are hard-wired in the code changes are more expensive than it is the case if they are stored in a file which is read by the system at run-time. They can also be stored in a database, but we do not want to access the database after an error already occurred, because in an error situation we probably cannot access the database anymore. It is necessary to read the file or database in advance. The error class interface hides the persistence mechanism used for error descriptions.
- *Structuring of error types.* If we want to react to an error in an appropriate manner, we have to classify errors more exactly. Just like a physician cannot give the same medicine to all of his patients. He has to make a diagnosis for a specific and effective medical treatment. With software it's the same. The error objects must give us as much information as possible, so that we can react in a well-directed way. There will be groups of errors which can be handled equally whereas others must be handled individually. To facilitate such a fine-grained treatment of errors we have to classify them. Such a classification helps to structure error situations. Either we add some classification attributes to the error class or we define new error (class) types and relate them to each other via inheritance ([Error Hierarchies](#)).
- *Generation of error classes.* In a big application we will get a large number of error types. It is sensible to use a standardized description for errors and to keep the error specifications in a project repository. Because error classes are not very complex, their description is not difficult and the implementation code for the error classes can be 100% generated.

Sample Code (Cobol)

The pattern is not restricted to object-oriented languages, the main ideas can also be transferred to 3GL languages. E.g. for a Cobol implementation the error class becomes data and operations in a module. This module is used by every application module and is generated from a textual specification. For instance, an error description consists of an identification number, a textual description and a number of types for the context parameters:

```
/* ----- errors: missing data ----- */
...
# 0815 aComment
   -- aDataStructureType
   1: aTypeForFirstElement
```

```

2: aTypeForSecondElement

# 0816 ...

```

Within an application module it is used in the following way:

```

*****
theOperationName section.
*****

...
*   >>> Error - Location 4 <<<
...

* -----
* „create error:“
* -----

move ERR-STATE of module-state of type-constants
      to module-state of global-variables
move 4 to error-loc of global-variables
move 0815 to error-id of global-variables
move 'theOperationName' to current-operation of global-variables

* -----
* „report error:“
* -----

...
move 0815 to error-nr of ...
move MODULE-SHORTNAME of .. to ...
move 'theOperationName' to operation of ...
move 4 to location of ...
*   supply values for the context parameters of the error message:
move ContextDataStructure to var1 of ...
...
perform error-prk-call
...
perform handle-exception
*   >>> End of Error - Location 4 <<<
...

```

This corresponds to the creation of an instance of an error class where the constructor immediately writes error information to the log. Reporting of the error 0815 is done according to the above error description.

Sample Code (C++)

The following code illustrates the interface of an exception base class in C++. It is very similar to the classes used in the **DATEV** projects as well as in the **HYPO** project [GKL95]:

```

class ExBase {
public:

    virtual ~ExBase( void );

    ExBase (
        const char* ExClass,
        const char* ExNumber,
        const char* ExText = "undefined",

```

```

        const char* MethodSignature ="undefined",
        const unsigned long LineNo = 0L,
        const char* ModuleName = "undefined",
        const char* DLLName = "undefined",
        const char* ProgramName = "undefined",
        const char* ErrorTime = "undefined",
        const char* UserAccount = "undefined"
    );

    ExBase ( const ExBase & eExcep );

    virtual ExBase* getCopy ( void ) const;
        // delivers a copy of this object
        // must be overwritten by each derived class
    virtual char* getExcepString ( void ) const ;
        // delivers a string that describes the exception
        // containing all stored instance information
    virtual char* getExClass ( void ) const ;
    virtual char* getExNumber ( void ) const ;
    virtual ExCategory getExCategory ( void ) const ;
    virtual char* getExText ( void ) const ;

private:

    ExBase & operator= (const ExBase &);
    ExBase();

    // instance variables

    char* ExClass;
        // unique identifier of an exception class
    char* ExNumber;
        // unique identifier of an exception relative in a class
    char* ExText;
        // situation specific description of the
        // exception situation
    char* MethodSignature;
        // contains signature of the method that raised
        // the exception
    unsigned long LineNo;
        // line number of the throw statement
    char* ModuleName;
        // filename the exception was raised in
    char* DLLName;
        // name of the DLL that raised the exception
    char* ProgramName;
        // name of the actual exe file
    char* ErrorTime;
        // machine time the error occurred at
    char* UserAccount;
        // user that raised the exception
};

```

The class is meant to store the characteristic information for an exception, so it contains little functionality and the implementation is straightforward. All data is implemented by the simple `char` type to reduce dependencies and internal error handling. All class arguments of `throw` and `catch` clauses are derived from `ExBase`.

Related Patterns

The pattern already suggests that there exist different groups or categories of errors. The pattern [Exception Hierarchy](#) deals with this topic and shows how to structure errors.

The pattern is also related to questions of error logging ([Backtrace](#)) and error abstraction ([Exception Abstraction](#)).

Known Uses

The exception handling concept of **Castek** [CSF96] is akin to this pattern and the Cobol example. Castek defines a common data area (exported by a root component which is the source of all other components) which contains data elements (all defined as text fields) like `Return_Code`, `Reason_Code`, `Severity_Indicator`, `Rollback_Indicator`, `Data_Modified_Indicator`, `Context_String`, and `Exception_Message`. They also define some services which operate on this data.

Within the **VisualWorks Smalltalk** environment exception handling is done by signals. For example there exists an `ErrorSignal` class. If a `raise` message is sent to a signal object it creates an exception object (of a predefined exception type). This object is automatically passed to the next matching signal handler which is then executed. It is possible to add a parameter object to an exception (e.g. an exception message). The „Frammento“ framework of the **DaRT** project is based on this environment.

The **Easy** project also uses an error class (although it is a more general exception handling class which they call return-code class). The class is common to all application classes and whenever an error occurs within a method an instance of this class is supplied with some error information.

The **Java** programming language offers a simple error class as part of the language class library. An error message (any string) can be passed to the constructor of that class.

Exception Hierarchy

Abstract

An Exception Hierarchy helps to structure a great number of errors and thus simplifies their administration as well as their usage by the application programmer.

Context

In a large project we have to deal with a great number of different error types which may be distinguished by criteria like severity (level) of an exception, handling strategy, frequency, context, cause of an exception, and location.

Without a defined structuring scheme a project will end in a mess of error definitions.

Object oriented languages support structuring of types via inheritance and allow definition of polymorphic error handlers.

Problem

How do you structure error types? What role does inheritance play in the structuring of errors?

Forces

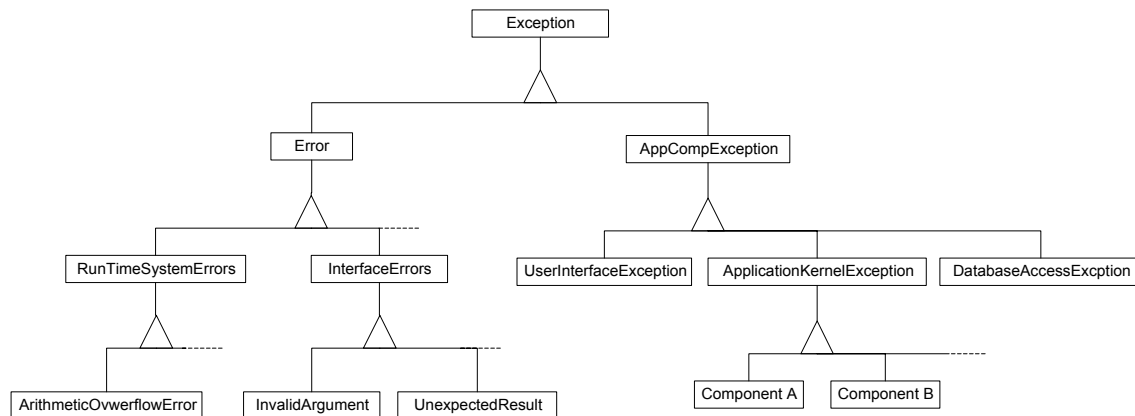
- *Classification granularity:* An inheritance hierarchy should be balanced: neither too deep nor too narrow. The more fine-grained the type structure the more specific error handling can be. The more specific the error types, the more complex the structure which increases the costs for implementation, maintenance and performance.
- *Structuring Criteria:* Structuring must be logical (following the same criteria). It must be clear to software developers where and how to insert new error types.
- *Polymorphism:* The hierarchy must reflect our needs for polymorphic error handlers.
- *Extendibility and reusability:* The structure must be extendible and adequate for different applications using the same error handling infrastructure.

Solution

The structuring of errors follows the structuring of the whole application. On a first level, this means to separate the following groups of errors: first, errors of the error handling component itself and second, errors which are specific to application components. Error handling as a common infrastructure offers types for basic (system) errors, which can occur in nearly every application component. These errors are generally not specified within every application component. Errors on the application side are specific to certain components and thus are also handled as explicit parts of the components. They can rely on the types of the error handling infrastructure and in contrast to common system errors they are an important part of the system specification.

Structure

For a system we arrange errors (or more general exceptions) according to the following hierarchy:



Each node represents a logical group of errors and a leaf is a concrete error type which is actually instantiated within the system. The grouping follows the question: where does such an error occurs in the system? Therefore, each node corresponds to a system component and the whole structure reflects the logical decomposition of the system. The `RunTimeSystemErrors` should indicate that also exceptions of components from the basic system environment (operating system facilities, libraries, ...) fit into this scheme.

Consequences

- *Granularity:* With a good system architecture the error hierarchy becomes really balanced and stable. The pattern does not necessarily lead to a fine-grained type structure. The designer can still decide up to which (component) level exceptions should be distinguished. This decision depends on the error handling strategy of the application.
- *Performance:* Because there is little functionality within the exceptions objects they cause no overhead for resolving dynamic method calls. The objects are very simple and can be generated.
- *Structuring Criteria:* The structuring criteria is very simple, so that the hierarchy is really intuitive and manageable.
- *Polymorphism:* The grouping allows us to write polymorphic error handlers. For instance, in Java or C++ we can write a catch-clause `catch(Error){..}` or `catch(DatabaseAccessException){...}` which automatically matches all classes contained in (derived from) this group of classes.
- *Extendibility:* It is not difficult to extend the hierarchy and to allow for different applications as long as their architectures match. If for each leaf in the hierarchy an exception (class) type can be defined within the namespace of the comprising component, no name clashes occur and the hierarchy also reflects the name space. This is a great advantage as we do not have to deal with global error identification numbers and their administration.

- *Reduced complexity*: A software developer's view on the hierarchy can be restricted to those error types derived from the class which corresponds to the component he is concerned with.

Implementation

Within object-oriented languages we can implement the hierarchy one-to-one. Every error defines a class (type). Due to inheritance and polymorphism we can easily catch all errors of a subtree. If a language does not offer this possibility we have to write a number of if-cascades or additional methods to match for the required exception types.

Even if the implementation language is not object-oriented, it is recommended to structure errors according to this pattern, at least during specification and design. Finally generators may be used to map this hierarchy to code in the implementation language. If it is not feasible to map each error type to a type in the implementation language, we have to imitate error types by means of other language features. Enumerations, strings, or simple numbers are possible alternatives for the definition of error type identifiers. Data and operations may collapse into one module.

For example the **TLR** project uses an enumeration to distinguish system errors and numbers for exceptions on the application side.

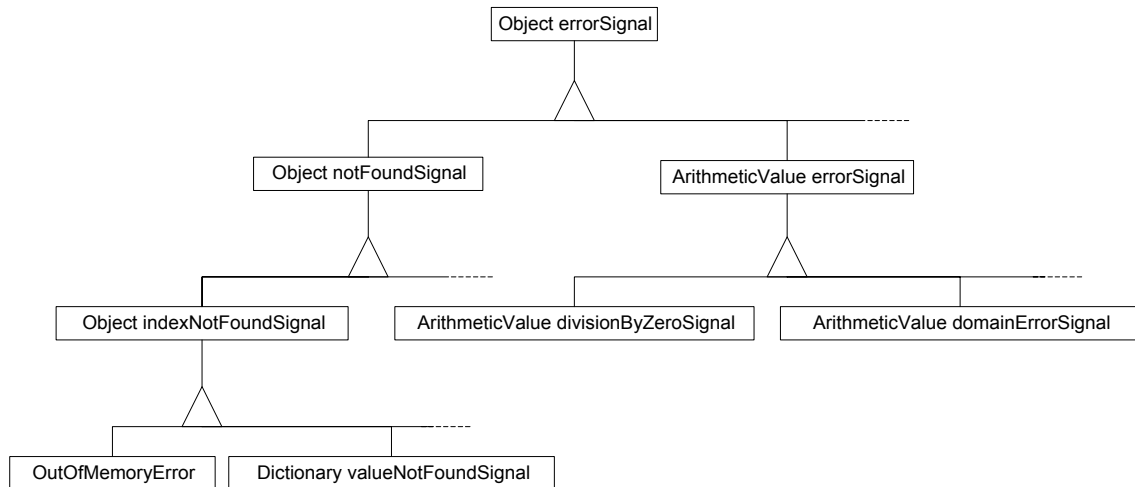
Different number domains for application components allow new error types to be introduced later on and the encoding may also reflect the hierarchy.

Related Patterns

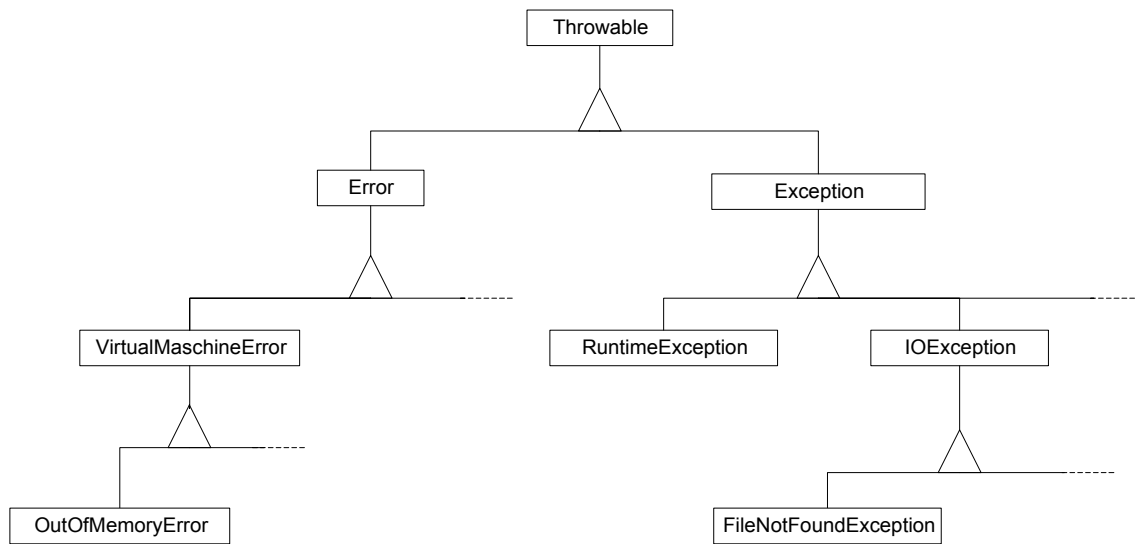
This pattern does not talk about the interior structure of an Error Object. The proposed error hierarchy facilitates Exception Abstraction.

Known Uses

In the **VisualWorks** Smalltalk environment, exception handling is done via signals. Signal objects exist for all important kinds of errors. They are class variables which can be accessed by special methods. These signals are also organized hierarchically (the diagram only shows a part of the hierarchy; an object name is followed by the method which returns the corresponding signal object):

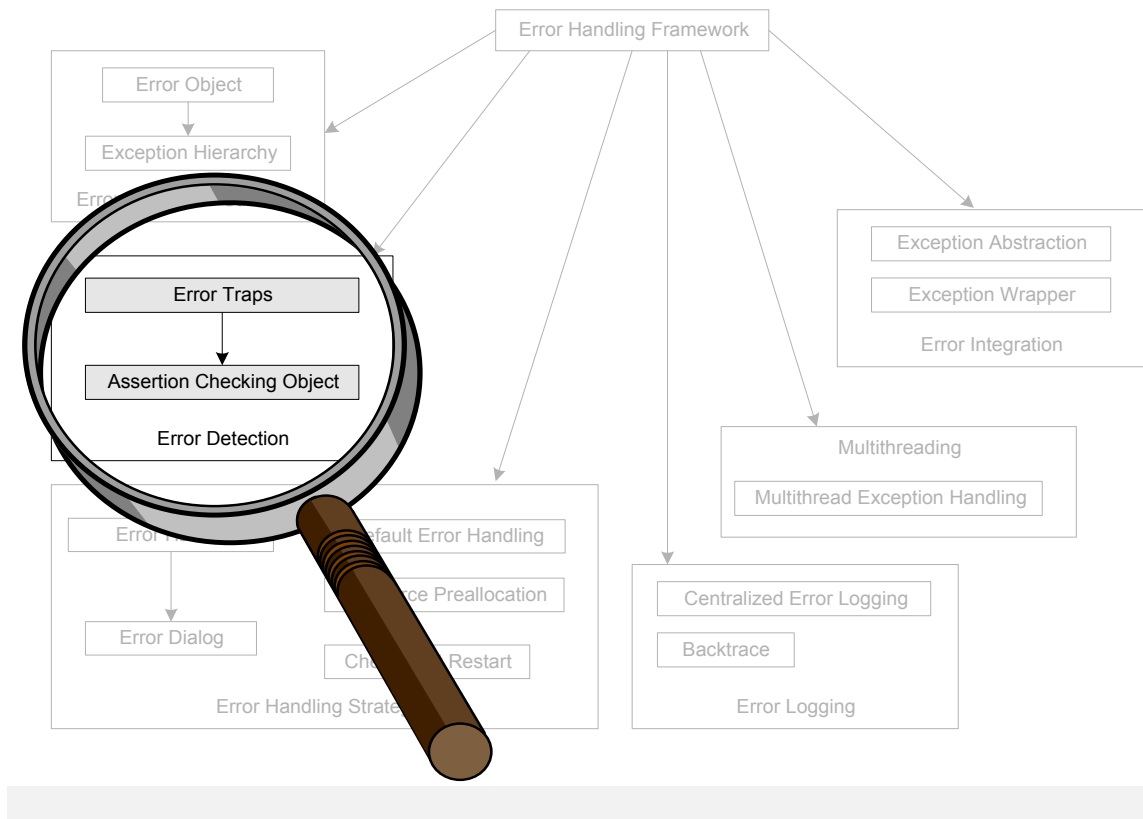


In **Java** the class `Throwable` is the root of the exception hierarchy. This base class is specialized into two different categories: `Error` for severe exceptional conditions and `Exception` for the use in application classes. The class `RuntimeException` is used for all kinds of basic exceptions which can occur anywhere in a program, so that they must not be declared within a method's interface:



The design of the **Hypo** project [GKL95], which is based on experiences from the **DATEV** projects, suggests an exception hierarchy according to this pattern. It also addresses the problem of different applications which should be based on the same exception handling and therefore must be integrated into the same overall error hierarchy.

2.3 Error Detection



Error Traps

Abstract

The pattern shows how to check for errors within a method.

Context

Before we can handle an error or failure we have to detect an error. For error detection we must enrich our code with a number of run-time checks, but a failure can only be detected in relation to a specification of the correct behaviour. The specification itself is assumed to be correct.

Problem

Which indicators are useful to detect erroneous situations and where should the traps be installed in the application code?

Forces

- *Complexity versus criticality*: You should relate necessary overhead (additional complexity, code size of source and executable) to the severity and frequency of errors and the size of the application code.
- *Performance*: On the one hand, we want minimal performance penalties, and, on the other hand, we want to be able to detect nearly all kinds of errors as soon as possible.
- *Robustness and consistency*: It is desirable to automate error checking as much as possible because automation supports a coherent design and correct implementation.
- *Maintainability*: To preserve maintainability of the application code, you should avoid cluttering of the code by a mass of error detectors.
- *Flexibility*: The possibility to activate and deactivate error detectors provides more flexibility.
- *Logging*: Error detectors need access to an error log to report detection events.

Solution

The idea is to verify the state and behaviour of the system against the specification at run-time by instrumentation of nearly every single method (except inline methods or macros). Thus we write a precise design specification of the classes in form of pre- and postconditions for every method and invariants. Every constraint must be made explicit.³

Then we check every method *m* in the implementation for the following error situations:

³ To enforce a particular constraint (e.g. a constraint on the relation between a group of objects) can be a difficult design task.

What?	Where?
Invalid parameters.	On entry of the method.
Violation of the pre-condition.	On entry of the method.
Unexpected results or failures of methods called by m.	Immediately after return of the method. If the language supports exception handling signaling of an exception automatically invokes an appropriate exception handler. Thus we do not need to instrument the code with an additional check for the result and the exception handler code can be well separated from the application code.
Violation of a method's invariant.	We have to distinguish between two kinds of invariants: some invariants are related to the whole class and they are checked at the end of every method of that particular class. This works well (the invariant holds on entry of every method) as long as the data of the class is only accessed and manipulated by these methods. Checking an invariant on entry as well as on termination of a method is necessary if an invariant concerns properties of shared classes (aliasing) or if an invariant is restricted to particular methods. In the latter case, the invariant can also be expressed as part of the pre- and postcondition.
Violation of the post-condition.	On termination of the method.

The list might be extended by individual checks for special assertions inserted by a developer (e.g. loop invariant). The pre- and postconditions are constraints about the class' internal state and the state of classes from the environment as formulated in the specification.

Structure

The following pseudo-code illustrates the structure of a method instrumented for error detection. We use the notation `[assertion ? action if the assertion is violated]` for a general error detector.

```

METHOD AnyMethod(aType1 aParam1, aType2 aParam2, ...) : aReturnType
BEGIN
    ----- error detection header -----
    [ aParam1 valid ? raise exception for invalid parameter ];
    [ aParam2 valid ? raise exception for invalid parameter ];
    [ invariant holds ? raise exception for violated invariant ];
    [ precondition holds ? raise exception for violated precondition ];

    ----- normal method body -----
    ...
    -- do something
    [ special test ? raise exception ];

    Result = aClass.OtherMethod(aValue);
    [ expected Result ? raise exception ];
    ...
    ----- error detection footer -----

```

```
[ invariant holds ? raise exception for violated invariant ];  
[ postcondition holds ? raise exception for violated postondition ];  
[ return value valid ? raise exception for invalid result value];  
RETURN aValue;
```

HANDLE

handle exceptions raised within the block

END

Consequences

- Whether this solution detects errors as early as possible depends on the method's size. The smaller the methods the higher the frequency of checking and thus detection is closer to the original cause of an error. The size of a method varies depending on the programming style and language. Roughly speaking methods in object oriented languages tend to be smaller than procedures in imperative languages.
- The implementation of the detectors is guided by the specification. As already mentioned in the context we assume that the specification is correct, so that this solution is not helpful to detect errors in the design specification. The question whether this solution is really effective and can detect a huge number of errors also depends on the quality (completeness) of the specification. If the specification carefully exposes the pre- and postconditions and invariants, a correct implementation according to this pattern will detect most implementation errors. The solution is not suited to detect loops (non-termination of a program).
A very critical situation is the incorrect implementation of an error detector itself. This danger is increased by the fact that pre- and postconditions as well as invariants are mostly hand-coded.
- Because the solution enriches the code of nearly every method there are strong effects on the performance of an application. Of course the kind of implementation influences the performance, but to really speed up the only choice is to switch off error detection. A compromise would be to restrict error detection to the critical parts of an application.

Implementation

- *Check methods.* By introducing additional methods to a class for checking particular properties (e.g. an invariant, the consistency of a class) we can avoid redundant code in a number of detectors which have to verify these properties.
- *Detector actions.* Once an error is detected a number of actions should be triggered: collection of context information, creation of an exception object, reporting the exception to the error log, cleaning up resources, trying to reach a consistent state and finally signaling the exception to the caller. Again, to avoid redundant code it is necessary to implement macros or methods for these actions.

- *Activation, Deactivation.* Introducing different checking levels and switches offers more flexibility concerning the intensity of checking. C and C++ programmers often use preprocessor directives (like `#define`, `#ifdef`) to implement compile-time switches (either a global switch via makefile or locally within the source files), which allows the complete removal of error detectors from the code. For a more fine-grained control, it is necessary to distinguish between different kinds of checks or to provide switches on a per class level. Consider who should be able to activate and deactivate detectors: is it sufficient to fix the error detection mode at compile-time by the software developers or do system administrators need run-time configuration capabilities ([Assertion Checking Object](#))?
- *Macros.* To implement error detectors with macros is very sensible. It helps to keep the code attachments small and readable, they prevent redundant code, are very flexible, easy to change or remove, compile time overhead is acceptable and run-time performance is very good. Note that it is also possible to use a preprocessor like that for C and C++ for other languages (e.g. Java, Cobol). Macros, however, have drawbacks for debugging as you cannot step into a macro with a debugger and macro expansion increases the code size.
- *Generation.* Generally, it is very helpful to generate as much code from the specification as possible. How much code for error detection might be automatically derived from the specification depends on the specification language, on the one hand and the programming language, on the other hand. But it is always possible to generate code templates, which must be completed by hand-coding. Especially the precondition and invariants are often specified by natural language, which makes it impossible to generate code for them. Otherwise, specification languages which support precise specification of preconditions and invariants by logic formulas (predicate calculus) also require to refine these constraints as long as they are expressed in an executable way.

Sample Code (C++)

If your exception log should contain some useful maintenance information, you need to include lots of parameters into an exception's constructor. Most of these parameters like `__LINE__` numbers or `__FILE__` names may be obtained automatically or may be generated using function calls like `_actualTime()`. Writing all these parameters by hand is far too expensive.

In C++ we can use parameterized macros that contain the minimum possible number of actual parameters. We can obtain all other information using macros like `__LINE__`, `__FILE__`, `__FUNCTION__` or whatever your development environment supports.

The actual macros you use depend on your project's requirements and programming environment. The following macros give an impression of what has been used successfully:

```
// check assertions
#define AssertTemplate(CONDITION, EXID, TEXT) \
    if ( !(CONDITION) ) \
        throw ExAssertionFailure(#EXID, #CONDITION, TEXT, __FUNCTION__, \
            __LINE__, __FILE__, __DLL_NAME__, __EXE_NAME__, __EXTIME__, \
            __USER_NAME__ )
```

```

#define AssertParam(CONDITION, TEXT) \
    AssertTemplate(CONDITION, EX_ILLEGAL_PARAM, TEXT)

#define AssertPrecond(CONDITION, TEXT) \
    AssertTemplate(CONDITION, EX_VIOLATED_PRECONDITION, TEXT)

#define AssertInvariant(CONDITION, TEXT) \
    AssertTemplate(CONDITION, EX_VIOLATED_PRECONDITION, TEXT)

#define AssertPostCond(CONDITION, TEXT) \
    AssertTemplate(CONDITION, EX_VIOLATED_POSTCONDITION, TEXT)

```

Other approaches in C++ use include files to insert the necessary code for error detection, implement check methods by templates, or use inline methods. The listed approaches can also be combined. Be careful using `__LINE__` and `__FILE__` macros within check methods. If they are expanded, the preprocessor will show you the source line of the check method and not the line in which the template was used. It is an advantage of the macro approach that we can hide `__LINE__` and `__FILE__` within the assert macros. Otherwise, every call of a check method within the application code must explicitly pass `__LINE__` and `__FILE__` as parameters.

Sample Code (Cobol)

We now look at code examples from the error handling of sd&m's **TLR** project [TLR95], which uses a standardized mechanism to perform result checks and type checking of variables. The following code excerpt (in an extended Cobol language) for a module operation illustrates the mechanism:

```

*****
OPERATION DoSomething
*****
PARAMETER
    in aParamID : aType
    ...
BEGIN
%CHECK SF (aParamID of $PARAMETER, aType) // checking the type of the parameter
    ...
%CALL ModuleName OtherOperation
    ( // parameter values...
      aKey )

%CHECK-RC (RC-OK, RC-NOK)
if $GRC = RC-NOK then
    %SET-DF (anErrorNumber, aKey)
end-if
    ...
%RETURN(RC-OK)
END-OPERATION

```

`%CHECK` is the type-checking command and `%CHECK-RC` compares a list of expected return-codes with the actual return-code. Deviation of the latter from the expected return-codes results in a system error.

From this „high-level“ Cobol code, pure Cobol is generated. Every module needs some types and variables for exception handling which are implemented by the following data structures:

```

* ----- types -----
01  type-constants
    ...
    05 rc-global
        10 RC-OK
            pic x(25) value 'RC-OK'.
        10 RC-NOK
            pic x(25) value 'RC-NOK'.
        ...
    05 module-state
        10 NORMAL-STATE
            pic x(2) value 'OK'.
        10 SE-STATE
            pic x(2) value 'SE'.
        ...
    05 exception-id
        10 SE-PARAMETER
            pic x(25) value 'SE-PARAMETER'.
        10 SE-UNEXPECTED-RC
            pic x(2) value 'SE-UNEXPECTED-RC'.
        10 SE-PRECONDITION
            pic x(2) value 'SE-PRECONDITION'.
        ...

* ----- internal module variables -----
01  internal-variables
    ...
    05 trace-buffer
        ...
    05 format-string
        pic x(80).
    ...

* ----- global variables -----
01  global-variables
    ...
    05 rc-global
        pic x(25).
    05 module-state
        pic x(1).
    05 current-operation
        pic x(25).
    05 exception-id
        pic x(25).
    05 exception-loc
        pic 9(3).
    ...

```

The type check command yields the following code:

```

* --- check type of aParamID ---
*
evaluate aParamID of ...
when ... continue
when ... continue
when other
    move aParamID ...
    move RC-NOK to rc-global of global-variables
*
    >>> System Error - Location 4 <<<
    move SE-STATE of module-state of type-constants
        to module-state of global-variables
    move 4 to exception-loc of global-variables
    move SE-PARAMETER to exception-id of global-variables
    move ...
    perform handle-exception
*
    >>> End of System Error - Location x <<<

```

```
end-evaluate
```

If the parameter value does not match the type constraints, a system error is produced. The `module-state` switches to `SE-STATE` (system error state) and an automatically generated sequence number (4 in this case) serves as an identifier for this error code. The identifier is written to the error log and helps to navigate back to the corresponding location in the source code. The constant `SE-PARAMETER` (defined within the data structures) describes the type of the error and is assigned to the variable `exception-id`. All the information is used by the routine `handle-exception` which finally writes the error log. `handle-exception` is a common routine of the exception handling component and is included in every generated Cobol module.

The command `%CHECK-RC(RC-OK, RC-NOK)` similarly expands to:

```
if not rc-global of global-variables = RC-OK and
  not rc-global of global-variables = RC-NOK
then
*   >>> System Error - Location 7 <<<
    ...
    move SE-STATE of module-state of type-constants
      to module-state of global-variables
    move 7 to exception-loc of global-variables
    move SE-UNEXPECTED-RC to exception-id of global-variables
    move rc-global of global-variables
      to trace-xparam of trc-buf-filed of internal-variables (1)
    move length of rc-global of global-variables
      to trace-length of trace-buffer of internal-variables (1)
    move 'The return-code is: %s` to format-string of internal-variables
    perform handle-exception
*   >>> End of System Error - Location 7 <<<
end-if
```

Variants

To reach higher performance, it is only considerable to instrument exported methods of a class.

Known Uses

This pattern is mainly influenced by the features of the **Eiffel** language [Mey88]. It contains `require` and `ensure` clauses to describe pre- and postconditions, supports the specification of invariants and also offers a `check` command to formulate assertions. To what extent these commands are actually executed at run-time can be configured within the compile environment. In the extreme case they are just documentations.

The **DaRT** project at sd&m uses assertions and implements a general `assert` method as part of their „Frammento“ framework ([Assertion Checking Object](#)).

Most C++ projects we know use special `assert` macros. There are examples in the **DATEV**, **HYPO** and **EASY** project.

As already mentioned the **TLR** project implements standardized error detection features into their sophisticated Cobol development environment. This environment is also used by and adapted to a number of other sd&m projects.

Further Reading

For detection of memory errors look for available tools on the market if the programming language or the environment does not support resource management very well. The book by D. A. Spuler [Spu94] contains useful tips and techniques for instrumentation of C and C++ code.

Assertion Checking Object

Abstract

Assertion Checking Object discusses an implementation of the Error Traps pattern in an object oriented language. It introduces a separate class for error detection and enables different levels of error checking by use of subtyping.

Context

We want to implement error detectors according to the pattern Error Traps in an object oriented language. We do not want to (or can't) use macros and the programming language offers no suitable instructions. Nevertheless, it should be possible to switch assertions on and off easily, because some checks are primarily used during development (loop invariants) and maintenance whereas others should be permanent (e.g. class invariants)

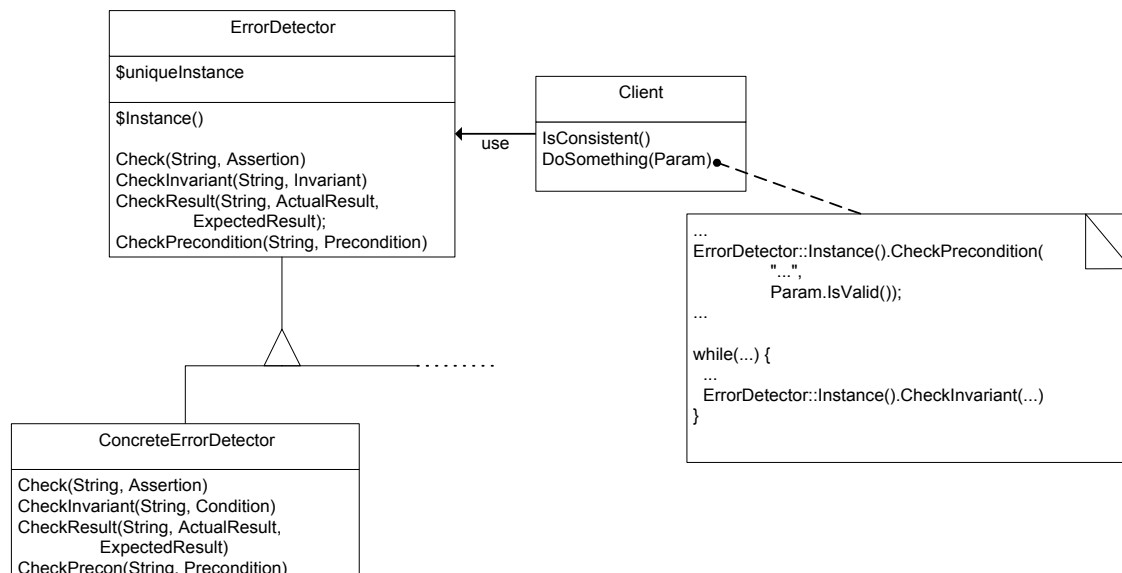
Problem

How do you implement the pattern Error Traps in this context?

Solution

Use a special class for error detection. The class offers methods to check whether a boolean expression yields true or false. In the latter case, the methods raise an exception and write information to the error log.

Structure



The error detector is a Singleton [GOF95]. By subclassing we can provide different kinds of error detectors (e.g. for different intensity of checking). Clients can use the `ErrorDetector` within their methods (e.g. to check consistency and invariants of classes).

Consequences

- By encapsulating various assert methods within one class it is easier to control and change the behaviour in case of violation.
- The singleton offers the flexibility to introduce subclasses or to change the class by providing more than one instance.
- The pattern offers the flexibility to adjust the check level on-line without recompilation.
- The maintenance and debugging of the code is easier and more transparent in contrast to macro processing. The compile-cycle can be more efficient.
- The `ErrorDetector` introduces an indirection which increases the communication and interaction of the classes. For every check a method call is necessary to get the current instance of `ErrorDetector` and another call to perform the actual check⁴.
- No redundant code (or no code at all) for level checking.

Implementation

- *Runtime Configuration:* To offer adjustable checking modes, the `ErrorDetector` may read the value of a corresponding configuration parameter at run-time. For instance, the configuration parameter can be an environment variable or an initialization file.
- *Macros:* We can still use macros (if possible) to simplify calls to `ErrorDetector`.

Sample Code

The following Java code illustrates an implementation. The base class `ErrorDetector` provides no functionality for the check methods and is the default initialization for the class variable which holds the instance of an `ErrorDetector`:

```
class ErrorDetector
{
    protected ErrorDetector() { instance = this; }

    protected static ErrorDetector instance = new ErrorDetector();
    public static ErrorDetector Instance() { return instance; }

    public void CheckPrecondition(String Descr, boolean Condition) {
        /* default: do nothing */ };
}
```

⁴ The call to the class method accessing the current instance might be optimized by the compiler (automatic conversion to an inline method).

```
}
```

Next, we define a class `FullErrorDetection` as a subclass of `ErrorDetector`, which implements all check methods:

```
class FullErrorDetection extends ErrorDetector
{
    public FullErrorDetection() { super(); }

    public static void On() { new FullErrorDetection(); }
    public static void Off() { instance = new ErrorDetector(); }

    public void CheckPrecondition(String Descr, boolean Precondition)
    {
        if (!Precondition)
        {
            throw new ErrViolatedPrecondition(Descr);
        }
    }
    ...
}
```

If the precondition is violated the method `CheckPrecondition` throws an exception `ErrViolatedPrecondition`, which can be implemented by

```
class ErrViolatedPrecondition extends Error
{
    public ErrViolatedPrecondition() { super(); }
    public ErrViolatedPrecondition(String s) {
        super(s); System.err.println(this.toString()); }
}
```

The constructor of this error class writes the information to the error log. In this simple example the standard output stream for errors (`System.err`) is used.

Known Uses

DaRT uses an assertion class (`FraAssert`⁵) in Smalltalk which provides a method `with:` to check a condition. For example

```
Assert that: ['Parameter is defined'] with: [aParam isDefined]
```

checks a parameter. The receiver (`Assert`) of the message is a global variable. The code behind the method can be switched by a class method `off`, which substitutes the class assigned to the variable `Assert` by a subclass (`FraNoAssert`) with an empty implementation.

Of course, in the client class remains a method call without any functionality which produces a run-time penalty. The following code shows the implementation of the class `FraAssert`:

⁵ The class is part of the „Frammento“ framework which was developed within the DaRT project. All class names in the framework have a prefix `Fra`.

```

FraSysDomainObject subclass: #FraAssert
...
that: aStringOrBlock with: aBlock
    aBlock value == true
        ifFalse: [ | tmpString |
            tmpString := aStringOrBlock isString
                ifTrue: [aStringOrBlock]
                ifFalse: [aStringOrBlock value].
            (self app msg: #Assertion) arg: tmpString; raiseSysError
        ]

!FraAssert class methodsFor: 'class initialization'!
initialize
    self on

!FraAssert class methodsFor: 'toggle'!
off
    Smalltalk at: #Assert put: FraNoAssert

on
    Smalltalk at: #Assert put: FraAssert

```

In case of an assertion violation, the method `that: with:` creates a new assertion message object (by use of the message identifier `#Assertion`). The first parameter of the method, which offers a description of the assertion, is passed to the message object as an argument. Finally, the exception is raised by the message `raiseSysError`. The code for `FraNoAssert` is straightforward:

```

FraAssert subclass: #FraNoAssert
...

!FraNoAssert class methodsFor: 'assertion'!

that: aString with: aBlock
    "Nothing is checked."

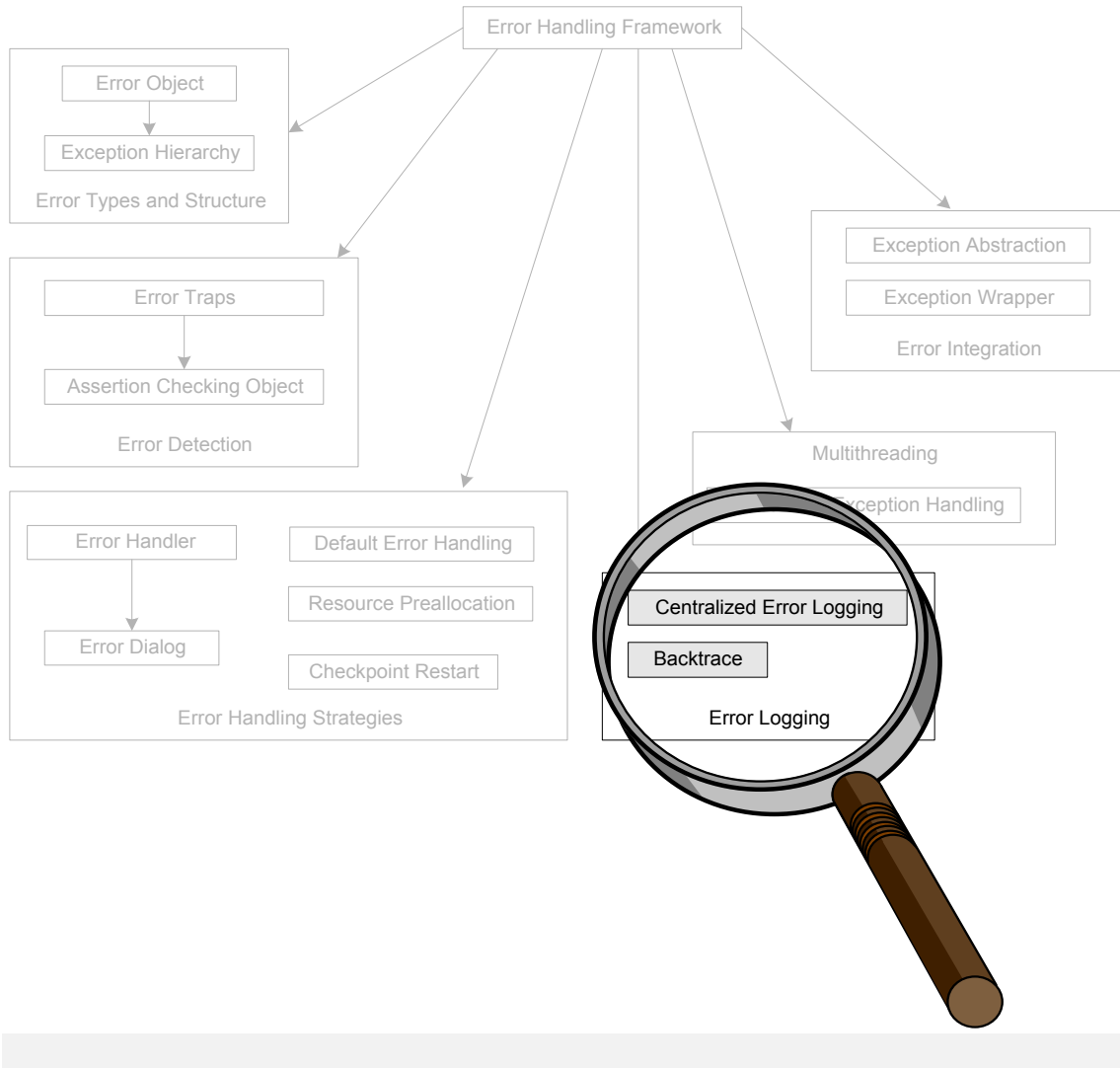
```

Related Patterns

This pattern offers a flexible and dynamic solution which fits best in the context of object-oriented languages and development environments. If you are looking for a solution more adequate for 3GLs study the samples given by the [Error Traps](#) pattern.

The implementation of different checking methods which can be switched dynamically is an application of the [Strategy](#) [GOF95] pattern.

2.4 Error Logging



Backtrace

Abstract

A Backtrace is a copy of an execution stack, which helps to examine an error situation. The pattern shows how to generate a Backtrace as part of an error log.

Context

We want to design the error handling for a large information system. In case of an error, the system should produce a suitable error log which helps to determine the cause of an error. The system environment (programming language, API) does not offer a suitable mechanism, so we are forced to implement one ourselves.

Problem

How do you collect and trace useful information that helps the system developers or the maintenance team analyze the error situation? Especially when we have no or limited access to the stack administered by the system itself.

Forces

- Because an error is an exceptional situation the performance costs for the backtrace should merely affect error handling but not normal operation.
- On the one hand, the information should be concise (no core dump of the whole system state), but, on the other hand, we do not know exactly what the most important information is.
- We do not want to program the code for *Backtrace* by hand. We want to use generators and macros instead. Therefore, the solution must be schematic.
- The solution must allow a correct and robust implementation since the error handling itself should not fail.
- It should not increase the size of the executable significantly.

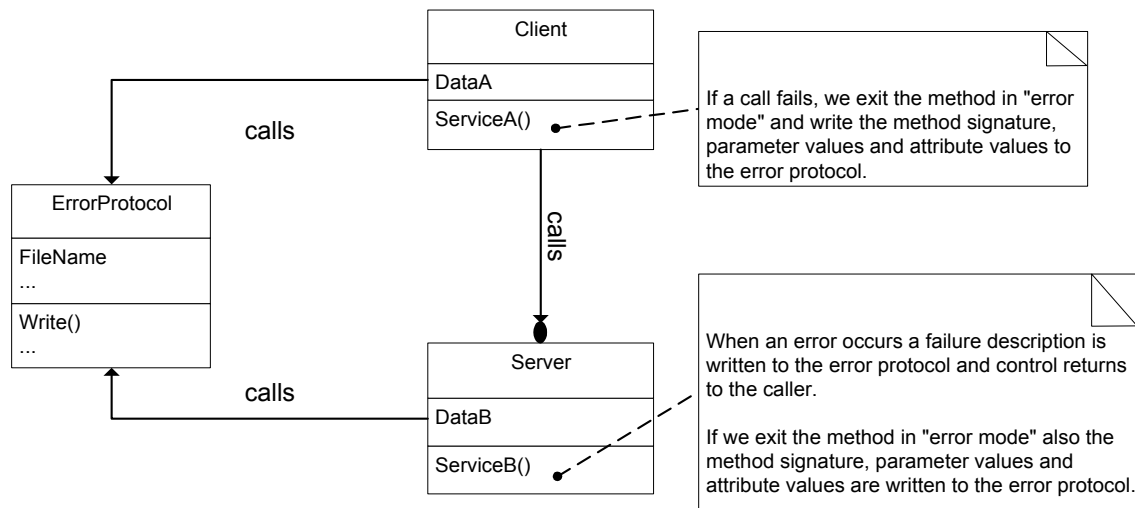
Solution

Once an error occurs within a method, the state of the method's class yields important information because it is possibly closest to the origin of the error. Moreover, we examine the current call-stack to get useful information about the history, which can also be important to analyze the origin of an error. If we have no access to the call-stack of the system, we construct the call-stack ourselves while the system is unwinding the call-stack. The states of all classes whose methods are stored on the call-stack are helpful. All this information is finally written to a log file, so that the content of this file reflects the genesis of the error.

Structure

The structure is very simple: We have clients which use services of other classes by calling methods of these classes. `Client` and `Server` are roles every class can play. A class can

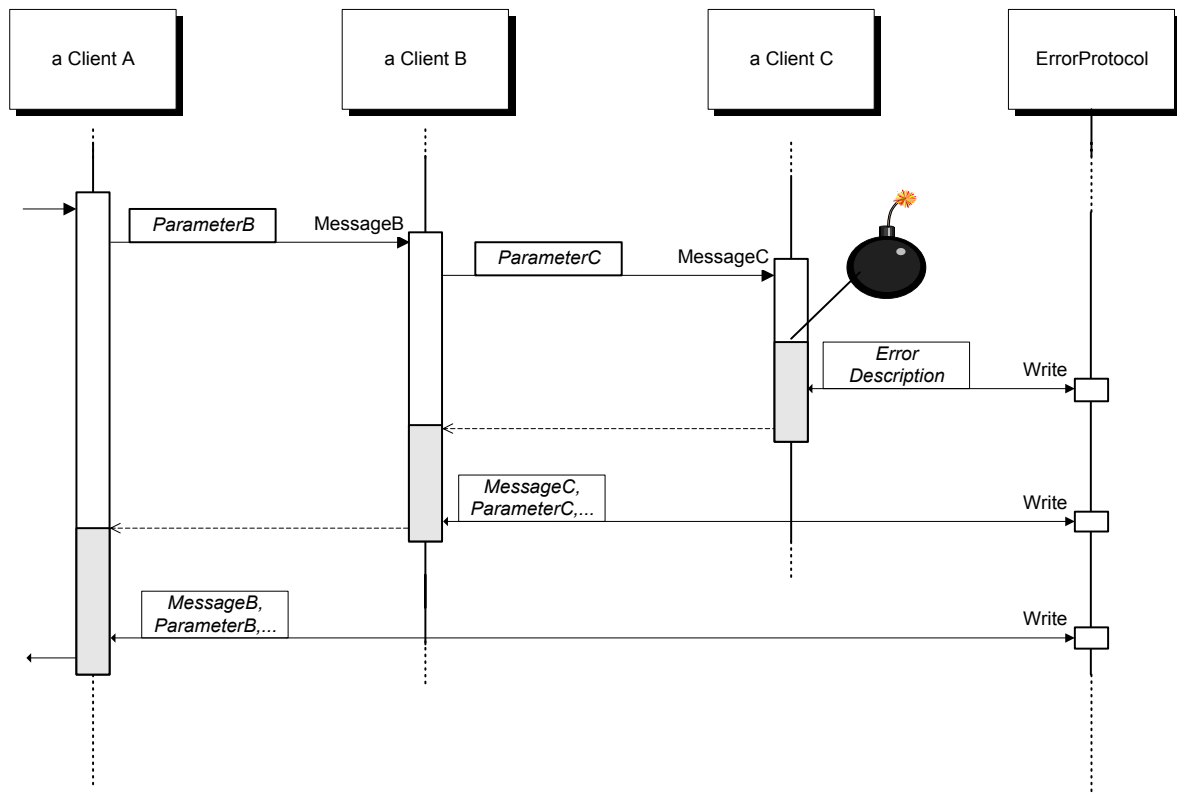
even call its own services. The `ErrorProtocol` is a Singleton, so that every `Client` or `Server` can access this class to write some data to the error log. Every method contains code to generate the backtrace once an error has been detected.



Dynamics

Because a method of an application class (as a `Client`) calls services of other classes (as `Server`) which themselves are clients of other server classes, a chain of calls is established. It is administered by a stack of the run-time environment. Whenever an error occurs, the task of the backtrace code is to generate a copy of this call-stack and to write it to the `ErrorProtocol`.

The following scenario illustrates the interaction:



Consequences

Instead of building a call-stack backwards, we can also build an own copy of the call-stack forwards. Every method call is traced on this stack and in an erroneous situation the stack is already available. Then, to build the stack can be part of general tracing facilities. The disadvantage of this approach is obvious: we permanently have to administrate the stack although an error seldom occurs. The Backtrace offers a solution to minimize the overhead during normal operation. But also the Backtrace is not for free: in 3GLs we always have to check for error-mode at the exit points of a method. In languages with exception handling mechanisms the situation gets better, the distinction is done by the system which is more effective and the code becomes clearer. The disadvantages of 3GLs can be partially compensated by advanced generation and macro processing tools [TLR95].

Implementation

- *Instantaneous Writing vs. Keeping a Stack.* In a simple implementation the `ERROR-PROTOCOL` writes all data directly to a file. In this case the first entry in the file is the top of the call-stack. But whether the top element would be the last or first entry would not be very important if appropriate viewing tools for the error log are available. Alternatively, the `ErrorProtocol` could store all error information in a stack internally and flushes the content to a file on request. Thus all information is still accessible and we can decide whether to dump the information to a file or to show the information to the user on the display. In case of recovery, we can delete the stack.
- *Collecting the information.* To fill the error log with useful information we need to know some data at run-time: the method signature, the line number, the name of the

class and maybe also the version and revision, the user name, process id and so on. There are generally three possibilities for getting the data:

1. *Statically*: At compile (preprocessing) time this information is directly included in the code (redundantly). Either special macros already exist or features of the compiler or self written generators are used.
2. *Dynamically*: The run-time system offers methods to get that information.
3. A third way is a combination of both techniques (dynamic access, no redundancy).

Sample Code (Cobol)

In sd&m's TLR project [TLR95] the generation of the system error log is hidden by the %CALL command of the extended Cobol language:

```
*****
OPERATION DoSomething
*****
PARAMETER
    in aParamID : aType
    ...
%CALL ModuleName OtherOperation
    ( // parameter values...
      aKey )
    ...
END-OPERATION
```

This call expands to:

```
* >>> CALL - Location 5 <<<
  move ... provide parameters
  move LOADMODULE_NAME of ModuleName of import-constants to
    loadmodule of global-variables
  move 5 to call-loc of global-variables
  move OtherOperation of ModuleName of import-constants to
    operation-name of header of ModuleName of import-interface
  perform OtherOperation-call
  ...
* >>> End of CALL - Location 5 <<<
```

The actual call is encapsulated by a separate section because the operation is used several times within the module:

```
*****
OtherOperation-call section.
*****
    ...
    call loadmodule of global-variables using ...
    end-call
    ...
    if exception-check of global-variables = FALSE
    then ...
    else
```

```

        move YES of true-false of type-constants to
            se-exit-flag of global-variables
        evaluate se-flag of header of ModuleName of import-interface
        when SE-STATE
*           >>> System Error - Location 7 <<<
            move SE-STATE of module-state of type-constants
                to module-state of global-variables
            move 7 to exception-loc of global-variables
            move SE-RECIEVED to exception-id of global-variables
            move ...
            perform handle-exception
*           >>> End of System Error - Location 7 <<<
            ...
        end-evaluate
    end-if
.
OtherOperation-exit.
    exit.

```

After the call of *OtherOperation* the variable *se-flag* signals whether an error occurred. Finally, we look at the routine *handle-exception*, which writes the call (name of the module, name of the operation, return code) to the log and exits the module:

```

*****
handle-exception section.
*****
    if call-trace of ... = TRUE
    and se-exit-flag of global-variables = YES
    then
        ... protocol operation call
        move ModuleName of ...
        move current-operation of ...
        move r-code of
            ...
        call ...
    end-if

    if se-exit-flag of global-variables = YES
    then
        ... exit module
        move module-state of global-variables to se-flag of ...
        goback
    end-if
    ...
handle-exception-exit.
    exit.

```

Known Uses

Most projects at sd&m use *Backtrace* for writing error logs.

The **DaRT** project uses Smalltalk at the client side and C++ on the server site, which results in different *Backtrace* solutions. The Smalltalk part displays a standard message to the user and asks the user for a file name. Then the system writes an error log to that file which con-

tains time and date, user login, an error description, and the call-stack. The latter is given by the run-time environment. For the C++ part they do not implement a backtrace but a forward-trace solution to avoid a standard catch block within every method.

At **TLR** the backtrace does not reflect the complete call-stack, only exported operations of a module are grasped.

In **Java** every throwable object (the class `Throwable` is the base class for all exception and error classes) automatically stores a string with the content of the call-stack during construction:

```
public class Throwable {

    private Object backtrace;
    private String detailMessage;

    /**
     * Constructs a new Throwable with the specified detail message.
     * The stack trace is automatically filled in.
     * @param message    the detailed message
     */
    public Throwable(String message) {
        fillInStackTrace();
        detailMessage = message;
    }
    ...

    /**
     * Prints the Throwable and the Throwable's stack trace.
     */
    public void printStackTrace() {
        System.err.println(this);
        printStackTrace0(System.err);
    }

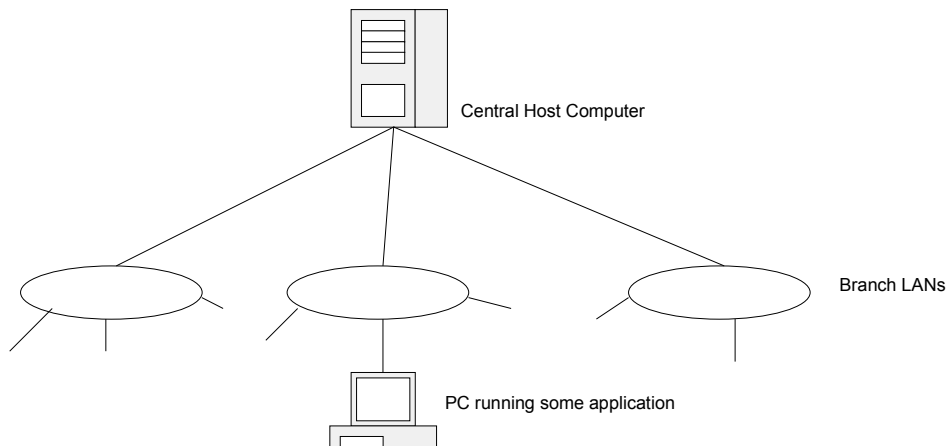
    public void printStackTrace(java.io.PrintStream s) {
        s.println(this);
        printStackTrace0(s);
    }

    private native void printStackTrace0(java.io.PrintStream s);
    public native Throwable fillInStackTrace();
}
```

Centralized Error Logging

Motivation

XY Bank has some 500 branch offices that run software on PCs. Each of these branch offices has an own local area network with up to 200 personal computers. For reasons of cost the help desk for the personal computers and the applications running on them is organized as a central function. You usually have one or more clerks per office but no maintenance personnel. Hence central maintenance personnel should have access to error information that is being produced by remote personal computers to be able to do remote maintenance for the PC's.



Context

You are the designer of the exception handling component for a large, distributed C/S system.

Forces

- On site service is expensive.
- Usually the bandwidth provided by a LAN is low compared to your needs. Additional modems are forbidden for reasons of network security. So logging into a users PC and getting core dumps via a WAN is impossible.
- Because of robustness and to avoid nested error situations a simple error logging is desirable.
- Maintenance personnel is not available on every branch office of a large company.

Problem

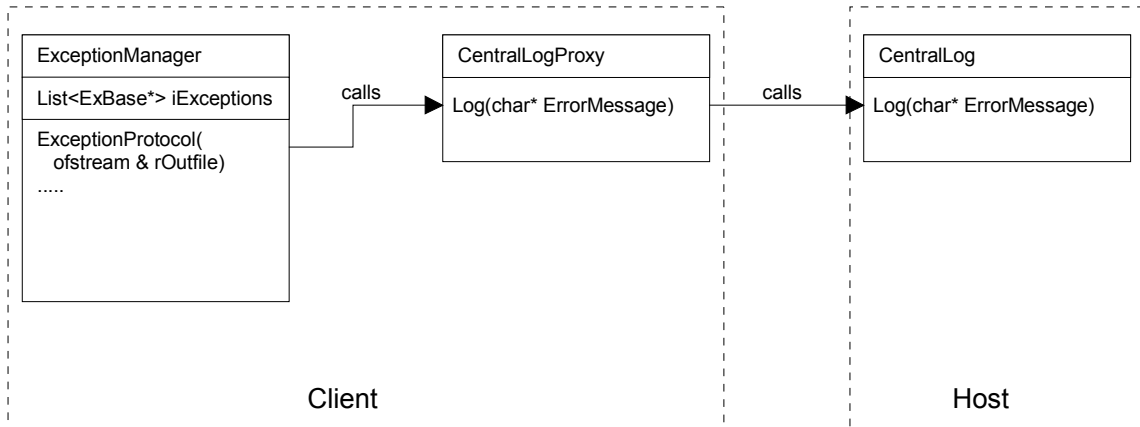
How do you organize exception reporting so that you can offer your maintenance personnel good enough information for analyzing the branch offices' problems?

Solution

Use a central error log on your host computer.

Structure

The structure is straightforward. The ExceptionManager uses a CentralLogProxy to log relevant ExceptionProtocols. The CentralLogProxy on the Client uses some communication mechanism like 3270 terminal emulation to propagate the ErrorMessage to the CentralLog.



Consequences

- All data is encapsulated by the class, which increases maintainability and avoids any global data.
- There are communication costs for sending errors to a central Host computer. On the other hand, the communication cost may be reduced by filtering exceptions for their severity level.
- The error logging becomes more complex and difficulties may arise from erroneous server communication.

Implementation

When implementing a central error log you should consider the following implementation issues:

- *INI-File for central error logging*: Central error logging can be made dependent on the severity of errors and on other factors like the application the error occurred in. To prevent the central error log from being flooded with error messages you should use parameters to be able to centrally steer error logging. These parameters may be represented in INI-Files.
- *Primitive Communications*: Some robust mechanism of communication should be used for error logging. Most WANs offer 3270 terminal emulation anyway. So it's a good idea to use normal terminal transactions for error logging environment.

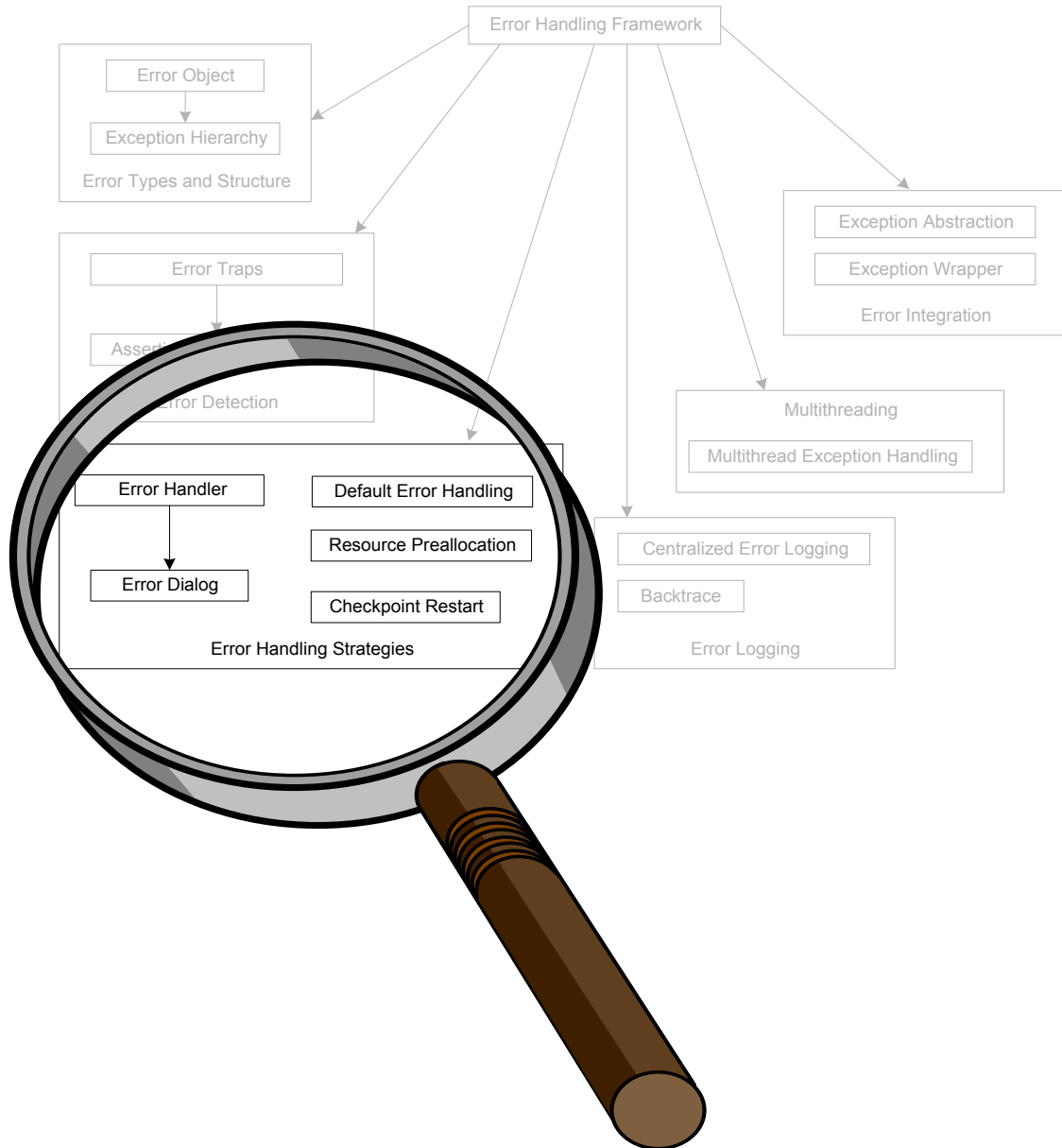
Known Uses

HYPO-Bank uses central error logging. This is described in the specification of „Exception Handling for C++ at HYPO Bank“ [GKL+95].

See Also

This pattern is an application of Proxy [GOF95].

2.5 Error Handling Strategies



Default Error Handling

Abstract

With a default error handler you can avoid unhandled exceptions. This is especially important because unexpected exceptions are the most serious ones.

Context

Within every method you have to think about possible exceptions of lower level services and how to handle them. Especially in the uppermost layers of the software it is important to handle all exceptions.

Problem

How do you ensure that you handle every possible exception correctly (no unhandled exception and limited damage)?

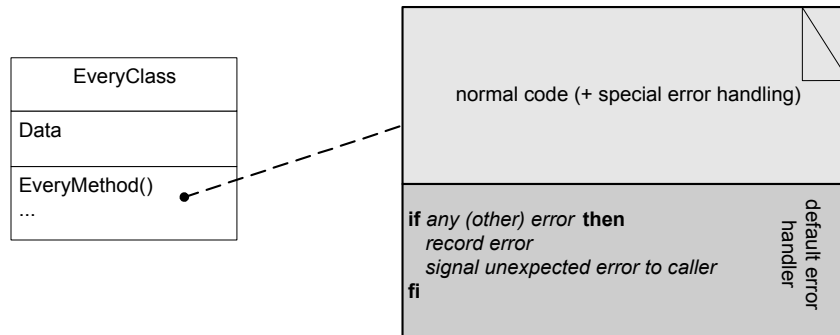
Forces

- We want to relieve the developer from writing similar error handling code for every method.
- We should not bother everybody with technical error handling stuff. Some developers want to concentrate on domain code.
- Individual error handling should be possible whenever necessary. For instance, a first version contains a very simple error handling which we want to refine in later versions.
- Error handling should be consistent to keep it maintainable.
- It is important to provide some error handling for unexpected exceptions.
- Changes to application code (e.g. new error detectors) or extensions should not result in change requirements for a large number of error handlers (e.g. all error handlers which are reachable by call paths of changed class methods).

Solution

Similar to the *default* section of a *switch* statement, we provide a default error handler for catching, logging and propagation of errors. The default error handler is unspecific and thus applies to every kind of error. It is added to every method where errors may arise.

Structure



Consequences

- Default error handling allows a safe and easy way of error handling. Since no special form of error handling is necessary, we do not need to think about error handling more deeply. We can automate default error handling so that nobody has to write redundant code by hand.
- The pattern can hide technical details concerning error handling.
- The pattern takes care that all exceptions are caught and recorded.
- Default error handling automatically supports a standardized and consistent error handling.
- As long as the default error handler is generic and its actions are not hard wired within various methods, the solution is robust against changes and extensions.
- The code size of the application increases.
- It might be more difficult to read the code and to understand the error handling.

Implementation

- Think of the places in your code to add default error handlers.
- Consider the actions of a default error handler.
- Use macros or generators to simplify and automate default error handling.

Sample Code

C++ supports default error handlers with a `catch(...)` clause, which matches every exception. We can use a macro that registers an exception with the exception chain and then throws another „empty“ exception further up. The `ExceptionHandler` uses the virtual `ExBase::getCopy()` method to copy any type of exception derived from `ExBase`.

```

#define _EX_STD_CATCH \
    catch (ExBase & anException) { \
        ExceptionHandler::Instance()->protocolException( anException ); \
        _EX_THROW( \
            ExBase,0, \
            "Propagate unexpected exception derived from ExBase", \
            anException.getExCategory() ); \
    } \
    catch (...) { \
        _EX_THROW(ExBase,0,"Unexpected Exception",ExBase::ExCat_Fatal); \
    }

```

Variants

You may also use a `throw` statement instead of

```

        _EX_THROW(ExBase,0,"Propagate ...", \
            anException.getExCategory()); \

```

This enables you to evaluate the exception further up. The above code was used in the HYPO project. The HYPO project's exception manager delivered the last meaningful exception (those with an error number not equal to zero) with his `getLastException()` method.

If the exception manager does not deliver the last meaningful exception and not the last exception, it is better to write:

```

#define _EX_STD_CATCH \
    catch (ExBase & anException) { \
        ExceptionManager::_Instance()->protocolException( anException ); \
        throw; \
    } \
    catch (...) { \
        EX_THROW(ExBase,0,"Catch All",ExBase::ExCat Fatal); \
    }

```

Known Uses

The pattern is used by the **Hypo** [GKL+95] and the extended Cobol language (developed for the **TLR** project [TLR95]) also provides default (system) error handling. **ANL** also applied the pattern in C++ and **CHAMPS** implements the pattern in Cool via return codes (by using a self written preprocessor). The CHAMPS implementation and the generation tools are very similar to those of the TLR project.

Another example is the **CLU** programming language which offers built-in default error handling.

Error Dialog

Context

You are writing an application program with an interactive user interface. The application kernel may result in exceptions we can not recover from automatically. Therefore, we have to inform the user about an error situation. Of course, the cheapest way to deal with exception handling is uncontrolled abort, but the results are perfectly confused users and overstressed developers who have no chance of finding the cause of program aborts. So we should invest a little to explain error situations to users and to give maintenance personnel the chance to react quickly to errors.

Problem

How to signal errors to an application user?

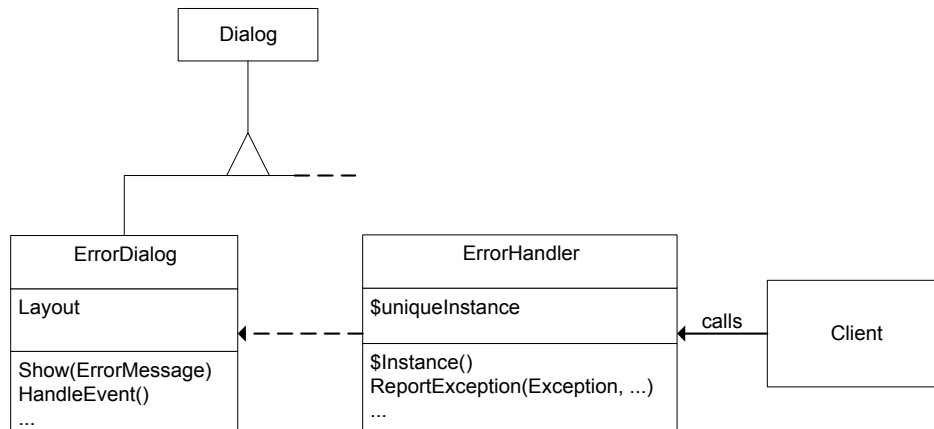
Forces

- *Adaptability*: The content (e.g. error messages) and layout of the error dialog boxes should not be hard-wired in the code, because it is likely to change.
- *Consistency and Style*: Error dialogs must be consistent and have to obey the particular interface style guides.
- *Clarity*: Error messages should be easy to understand and should give necessary information to the user.
- *Decoupling*: The application kernel and the basic error handling infrastructure should be totally decoupled from the error dialog.
- *Simplicity*: A small number of error messages and dialogs reduces complexity and eases maintenance and consistency.
- *Flexibility*: Some errors must be reported to the user immediately whereas others might be collected first and shown to the user once as a list. Within the application user interface, not only errors of the application kernel must be handled but also internal errors of the interface.

Solution

We use a special Error Handler which offers a report method to signal any exception to a system user. The error handler opens a dialog and displays an error message, which is available from the error object.

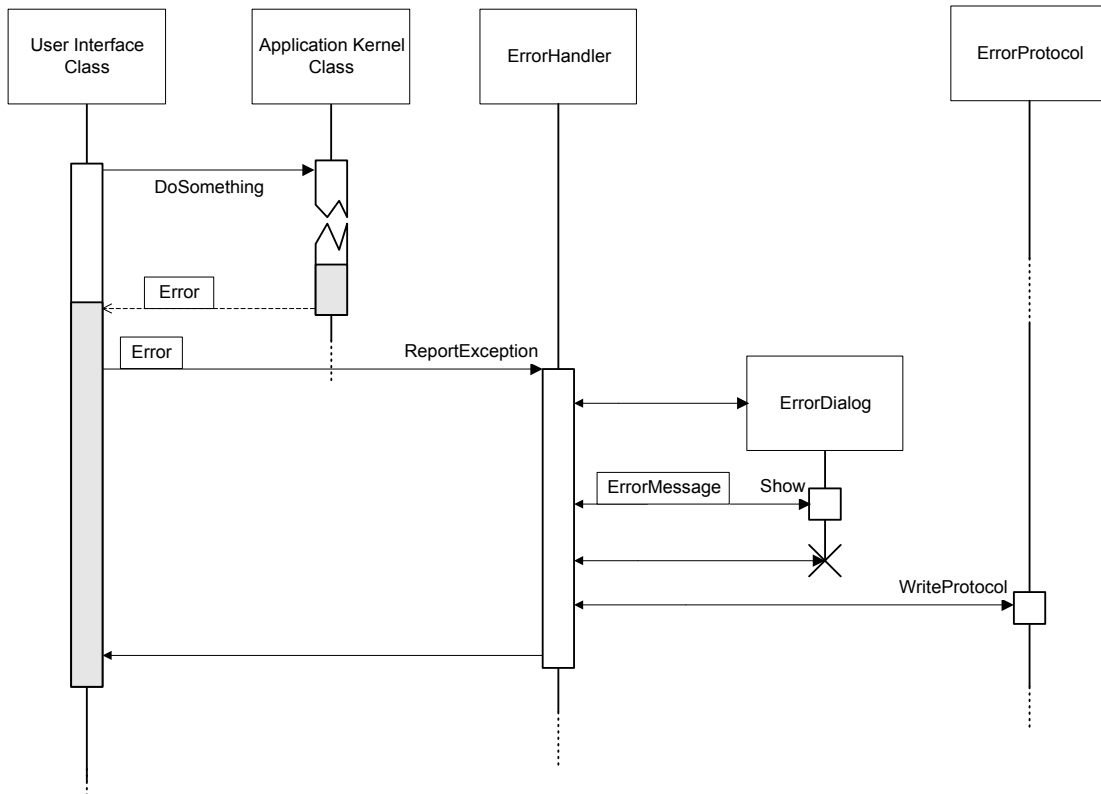
Structure



Participants

- **ErrorHandler**
 - This class is a Singleton [GOF95].
 - Every class within the application's user interface can use an `ErrorHandler` to report exceptions to a user.
 - An `ErrorHandler` does not know about the representation of errors on the user interface. It creates an `ErrorDialog` instead and calls `Show` to display an error message on a screen.
- **ErrorDialog**
 - The `ErrorDialog` is responsible for displaying error messages on the screen.
 - Generally, this dialog will be a modal dialog.
 - The `ErrorDialog` uses lower level services (e.g. GUI-framework).
 - To get the right message for an error the class uses `CreateErrorMsg` of `ErrorMsgMapping`.
 - `ErrorDialog` may inherit from a `Dialog` base class offered by a GUI-framework.
- **Client**
 - The client is a control object within the user interface. It uses the `ErrorHandler` to display a message for an error which was signaled by another method.

Dynamics



Consequences

- The user is informed about a program abort and is not confronted with an unmotivated abort or some cryptic error messages.
- The `ErrorHandler` can be used by various user interface classes (especially if the application uses several windows). Hence, only one place exists to control where to get the correct dialog class and how to display an exception.
- A separate `ErrorHandler` allows to restrict its usage to the user interface layer, so that no class from the application kernel can call `ReportException`.
- The pattern is not restricted to a graphical user interface, it applies as well to a textual interface.
- We can extend the `ErrorHandler` for further services like Exception Abstraction or error buffering.
- The maintenance personnel are able to track errors faster with the information provided by the Backtrace that may be found in error log files.

Implementation

- *Error Messages.* The information which is shown to the user is given by the message texts of Error Object. Such a message can be based on a parameterized text template,

which is instantiated by the error object. The error object also knows where to find the text template. If we want to change the text of an error message, the location of the template matters: With an error message file or an error message database, it becomes possible to change the messages without recompilation.

- *Error Abstraction.* We probably do not want to display errors as they come up from the application kernel. It is often desirable to provide an additional mapping between errors signaled by the application kernel and those shown to the user. Therefore the user interface extends the Exception Hierarchy by new Error Object. The error mapping is an Exception Abstraction.
- *Flexibility.* To get more flexibility an abstract base class for the `ErrorDialog` can be defined.
- *Different GUI-frameworks.* We can improve portability by creating an `ErrorDialog` via a Factory [GOF95].
- When implementing the scheme in a framework based GUI-application it might be hard to have one spot where all exceptions come to surface. Some frameworks allow definition of a central error handler, some others might force you to instrument each callback with a separate call to the error handler:

```
void SomeGUIClass::SomeCallback (Event& Msg)
{
    try {
        ApplicationKernel.doSomething();
    }
    catch(...) {
        // call the central error handler here!
    }
};
```

Sample Code (C++)

An `ExceptionDialog` contains all information necessary for displaying the user messages, e.g. window size, colours, buttons, window title. If different views of an exception are possible, different `ExceptionDialog` classes might exist. Alternative views can be implemented by providing an abstract base class for exception views with the concrete views derived from it.

```
class ExceptionDialog : public SomeDialogBaseClass {
...
public:
    ...
    ExceptionDialog();
    virtual ~ExceptionDialog();

    void Show(char* MessageText);
    ...
private:
    ...
};
```

```
}
```

Such a base class might be provided by a user interface framework. The method `DisplayExcep` is called by the `ExceptionHandler`, which implements the link between the `ExceptionHandler` and the `ExceptionDialog`:

```
class ExceptionHandler {
public:
    ...
    void ReportException(const ExcepBasis & Exception);
    void ReportWithoutProtocol(const ExcepBasis & Exception);
    ...

private:
    ExceptionDialog aDialog;
}
```

If an exception is caught within the user interface and the user should be informed about the exception the `ExceptionHandler` is used: the method `ReportException` may be called and the exception object is passed as a parameter. Of course, the class must be able to access an `ExceptionHandler` object. Somebody must create an `ExceptionView` and then an `ExceptionHandler` which gets a reference to the view object. The implementation of the `ReportException` method may look like:

```
void ReportException(const ExcepBasis & Exception)
{
    ...
    char* MsgText;
    unsigned short &Len;
    ExceptionDialog aDialog();

    Exception.GetMessageText(MsgText, Len);
    ...
    aDialog.Show(MsgText);
    ExceptionProtocol::GenerateProtocol();
    ExceptionProtocol::DeleteExceptions();
    ...
}
```

In this example only the message text of the last exception within the exception stack is shown to the user. The other exceptions and the technical information are written to the log. Another view could present the whole stack to the user. This is a question of user interface design.

To complete the example, we now use the `ExceptionHandler` within our main function:

```
void main( void ) {

    // initialize an exception protocol file for later use
    ofstream ExProtocolFile("ExProt.txt");
    // initialize exception handler for later use
    ExceptionHandler::Initialize();
    ...
}
```

```

try{
    mainLoop();
    cout << "regular end of program" << endl;
}
catch (ExBase & anException) {
    ExceptionHandler::Instance()->ReportException(anException);
}
catch (...)
    ...
}
}; // end main

```

A typical non-graphical dialog for reporting errors is:

```

The program will be terminated due to a severe system error.
The message is:
Database disconnected or not ready
Please inform an operator.

```

The above may as well be displayed in a dialog box if you have a GUI system. The user may have a look at the exception chain if he or she is interested.

Sample (Smalltalk)

We now take a look at code fragments for a (VisualWorks) Smalltalk implementation of exception handling (which is mostly taken from the DaRT project with some minor adaptations).

In the user interface layer, there is an exception handler as a part of a so called application model which e.g. controls code including calls to some methods of the application kernel:

```

self excepthdl do: [
    "this is the code-block which is guarded by the exception handler"
    ...
].

```

The exception handling is activated by sending a `do` message to the exception handler. This message requires the code which should be protected by the handler as an argument. This corresponds to a `try`-block in C++.

An `ExceptionHandler` object knows an `ExceptionHandlerDialog` object which is able to display an appropriate message. This object is accessed by `self displayObject`. This method gets an `ExcepInfo` object as a parameter, it is available via the parameter method of the exception object (`tmpException parameter`). Thus it is no problem to change the `displayObject` in an exception handler to provide different representations for exception messages.

To get an impression of `FraExcepDialog` look at the following code

```

FraSysApplicationModel subclass: #FraExcepDialog
...

```

```

show: anFraExcepInfo
    ...
    self openAsDialogInterface: self msgSpec.

msgSpec
    "UIPainter new openOnClass: self andSelector: #msgSpec"

    <resource: #canvas>
    ...

```

It contains the detailed specification (`msgSpec`) for a message window. The exception information which have to be printed within this window are encapsulated in a `FraExcepInfo` object which is a parameter of the `show` message:

```

FraSysApplicationModel subclass: #FraExcepInfo

arg: anObject
    args add: anObject!

args
    ^args isNil
        ifTrue: [^#()]
        ifFalse: [args]!

extraString
    ^extraString!

extraString: aString
    extraString := aString!

userMsg
    "returns the full string for the user message with all args
    inserted"

raiseSysError
    ^FraExceptionHandler sysErrorSignal raiseWith: self errorString: ' ' !

```

The actual message information is encapsulated by another class (`FraMessage`) which is used by `FraExcepInfo` internally. By `userMsg` the complete message string is composed by the text from `FraMessage`, the possible arguments and the `extraString`. To replace the placeholders in a message text by the arguments stored in `args` the `FraMessage` offers a method `textWithArgs`:

```

FraSysDomainObject subclass: #FraMessage

id
    ^id

id: aSymbol
    id := aSymbol

catalog
    ^catalog

catalog: aSymbol

```

```
catalog := aSymbol

text
  ^text!

text: aString
  text := aString
...
textWithArgs: anArray extraString: aString
  "The message text and aString are composed and the formal
  parameters %1, %2 ... are replaced by the values in anArray.
  Returns nil, if no message text is available."
...
readCatalog: aSymbol from: aString for: anAppSymbol
  "aString is the name of a file containing the messages for the
  catalog aSymbol. The messages are written to the dictionary
  for the application anAppSymbol"
...
```

With readCatalog the messages are read from a file into a dictionary of the application image. This class is not specific to exception handling, it can be used for all kind of messages.

Variants

Instead of creating an ErrorDialog within ReportException we can create one instance within the constructor and store the reference as an attribute. Then the dialog is only enabled and disabled by ReportException.

Known Uses

DATEV's IDVS Applications use the above error reporting scheme. The scheme may also be found in the Exception Handling concept that was developed for **HYPO**-Bank [GKL+95].

Another project – already mentioned in the Sample Code section – is the **DaRT** project.

Checkpoint Restart

Context

You are writing an application program with a batch interface for off-line processing. The application kernel may result in errors which must be handled by the batch interface. Computing resources on a mainframe can be short and expensive.

Problem

How do you avoid a complete rerun of a batch as a result of an error?

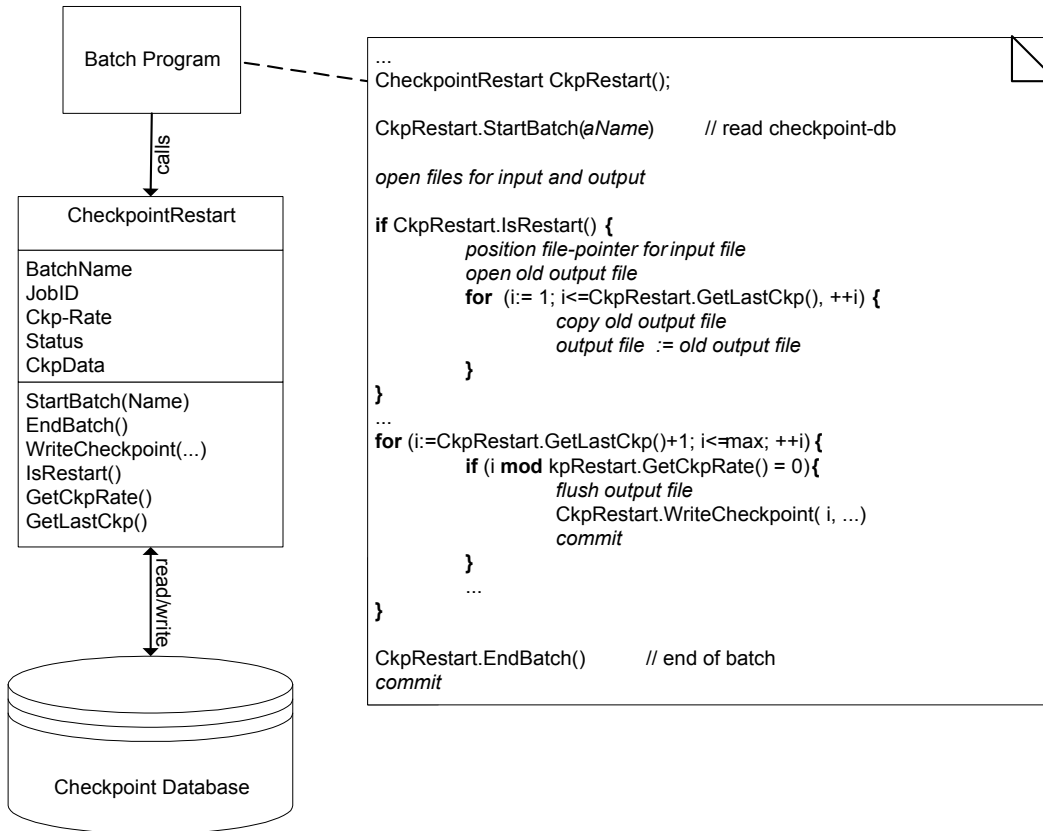
Forces

- The application kernel should be independent from the design of batch programs. Error handling of the application kernel should work with an interactive interface as well as with a batch interface.
- Although we continue processing, the user of the batch has to get a log of all error situations.
- Batches are long-running jobs (mostly at night). It is not feasible to stop a batch whenever an error is signaled.
- Often there is not a single, isolated batch but a complicated network of batches with various start dependencies.

Solution

Treat the whole batch as a nested, long-running transaction and implement a checkpoint-restart mechanism for error recovery. It saves consistent processing states at certain intervals (*checkpoints*), where to restart from in case of an abort. The checkpoint mechanism uses a special database for administrative data. The number of processing steps per interval (*checkpoint rate*) is configurable.

Structure



Participants

- **Batch Program**
 - Represents a general batch program.
 - Is client of CheckpointRestart.
 - Reads data from the checkpoint database with `StartBatch()` at the beginning of a batch.
 - Checks for a restart situation and looks for the correct input and output files.
 - Saves the current processing state (`WriteCheckpoint`) depending on the checkpoint rate.
- **CheckpointRestart**
 - Controls the checkpoint-restart mechanism.
 - Is responsible for the persistence of checkpoint data.
- **Checkpoint Database**

- Contains the data necessary to restore a consistent system state at restart (e.g. counter values, file locations).

Dynamics

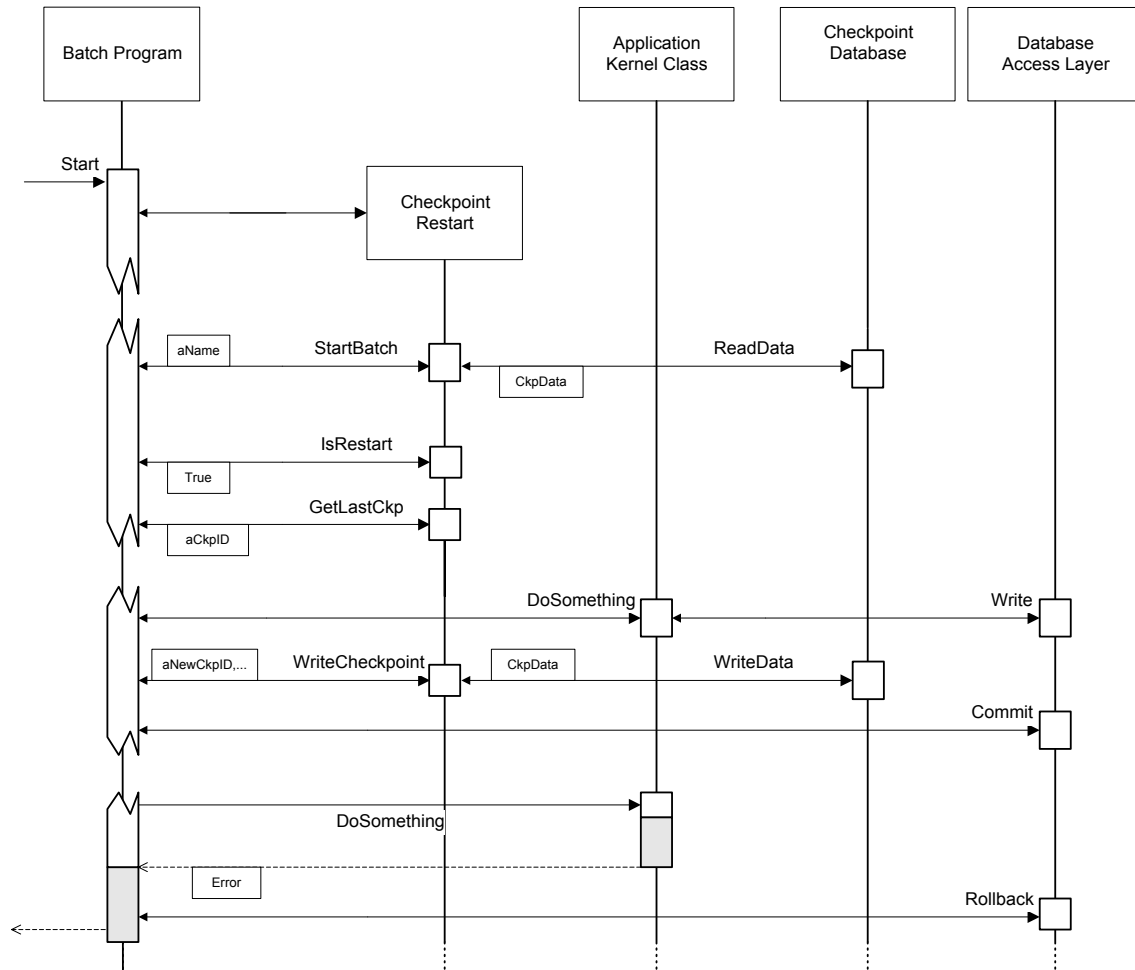


Figure 13: The scenario shows a batch which is aborted due to a serious error. The batch uses the Checkpoint Restart class to write checkpoints, so that a restart is possible.

Consequences

- Checkpoint Restart allows an early unlocking of database tables. A single commit at the end of a batch unnecessarily blocks a large number of tables.
- The pattern decreases performance, therefore it is important to choose a reasonable checkpoint rate.
- The batches becomes more complex.
- The pattern requires additional storage resources (files, checkpoint database).

- If a batch program is based on application kernel objects, we possibly have to store the object's states (partially) in the checkpoint database. In this context the pattern may lead to changes in the classes of the application kernel (to access the necessary data). Those additional dependencies negatively affects maintainability of the application.

Implementation

- Take care to correctly initialize the checkpoint data when the batch starts the first time.
- Determine a suitable checkpoint rate.
- We can sequentially read over the correctly processed records to position the input file pointer. Depending on the file system a direct positioning might also be possible.
- In host environments (e.g. MVS) special job control languages (JCL) exist to start and control batches.
- Consider whether an operator controls the restart of a batch or some automatism exists.

Variants

Instead of copying the output file in case of restart, we can also continue writing to the output file and remove doublets by sorting afterwards. Of course we have to ensure that normally no doublets occur. Another alternative copies the output file whenever a checkpoint is written. Then we can immediately continue with the copy (backup) of the output file at restart.

Known Uses

Checkpoint Restart has been successfully used in a number of large projects at sd&m (e.g. **TIAS**). The mechanism is well-known from database systems or more general transaction processing. The **INS** project (installation of the IDVS for DATEV) uses Checkpoint Restart for a long running server installation process.

See Also

The pattern was motivated by the sd&m lecture notes about Batch-Design [Sur96].

Error Handler

Context

Within components we either detect errors or errors are signaled by services of other components. We have to write error handling code to react to these errors.

Problem

Where and how do you handle errors?

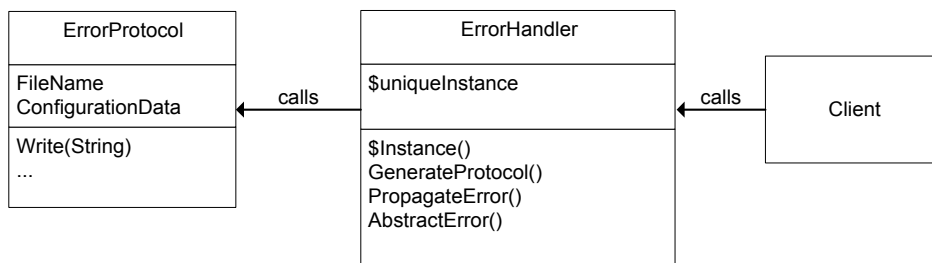
Forces

- Every error must be finally handled.
- For propagation of errors to an application user we need adequate error information.
- The error log must be written.
- Error handling must be consistent (different incompatible error handling strategies, interruption of error propagation).

Solution

We encapsulate error handling services within an `ErrorHandler` class. This class is a Facade [GOF95] for the whole error handling subsystem and as a Singleton [GOF95] it is accessible for every application component.

Structure



Consequences

- The pattern helps to structure error handling and prevents that nearly every method is cluttered by some error handling code.
- By centralizing error handling functionality within one class, it avoids redundant code and supports consistency.
- We are able to exchange error handling functionality without any changes to the application code.

Implementation

- *Error Buffering.* The ErrorHandler may store a collection of error objects.
- *Programming language features.* Implementations of ErrorHandler vary within different programming languages depending on the language features. In C++ and Java a catch-block contains error handling code. In Smalltalk such a code block is given to an error object as a message parameter. Again, other languages do not support writing error handlers at all. Error handlers may be modules or classes or simple functions.
- *Handling unexpected exceptions.* We have to expect the unexpected to get a robust system. Implementing a Default Error Handling is a common strategy to solve this problem.
- *Where to place error handlers?* We at least need error handlers in the uppermost level of an application (user interface part, main function) to inform users or to write error information. In an interactive application we need an Error Dialog.
- *Backtrace.* An ErrorHandler can be responsible for the generation of a Backtrace.

Sample Code (Smalltalk)

In the presentation layer there is an exception handler as a part of a so called application model which e.g. controls some application code:

```
self excepthdl do: [  
  "this is the code-block which is guarded by the exception handler"  
  ...  
].
```

The exception handler is accessed via a message `excepthdl` from the application model. This means that there is some initialization which creates a new `ExceptionHandler` object and configures and installs it for the application model. The configuration is done by the method `monitorExcepMessage`. The exception handling is activated by sending a `do` message to the exception handler. This message requires the code which should be protected by the handler as an argument. This corresponds to a try-block in C++.

To understand what the exception handler really does we now look at the `monitorExcepMessage` and `do` method of the class `ExceptionHandler`.

```
do: aBlock  
  Object errorSignal  
  handle: [:ex |  
    ex signal = self monitorSignal  
    ifTrue: [self handleBlock value: ex]  
    ifFalse: [  
      self unwindBlock value: self appMdl.  
      ex reject ]  
  ]  
  do: [self cursor showWhile: aBlock]
```

This method executes the code given by the parameter `aBlock` (see last line) and shows a certain cursor while executing the block. Which cursor is shown to the user is defined dynamically by calling the `get`-method for the instance variable `cursor`. It returns the necessary code. The cursor is only one example for the configuration possibilities of this method. In the `handle` block there are three other instance variables (all the methods called for dynamic configuration are shown in italics): `monitorSignal`, `handleBlock` and `unwindBlock`. These variables can be set by the abovementioned method `monitorExceptionMessage`. But before we look at code for this method, we still have to explain the code of the `handle` block in the `do` method above. First the exception handler checks whether the signal of the exception equals the signal which we want to catch. This signal is defined by the instance variable `monitorSignal`. If the comparison yields true, the actual code for handling the exception is executed, which is stored in an instance variable again and can be accessed by `handleBlock`. The exception object is passed as a parameter. If an exception occurs that does not match the signal, there must be other handlers which react to that exception. The block is unwinded and the exception rejected.

Now `monitorExceptionMessage` can be defined by:

```
monitorExceptionMessage
    "configure the signal which should be detected by this handler"
    self monitorSignal: SysErrorSignal.

    "the exception message is shown to the user"
    self handleBlock: [:tmpException | | tmpReturnValue |
        tmpReturnValue :=
            self displayObject show: tmpException parameter.
            tmpException willProceed
            ifTrue: [tmpException proceedWith: tmpReturnValue]
    ].

    "try to close the model:"
    self unwindBlock: [:tmpAppMdl | tmpAppMdl closeRequest].

    "show waitCursor during execution"
    self cursor: Cursor wait.
```

Variants

Instead of a single `ErrorHandler` we can also instantiate `ErrorHandler` within every method of an application class or we generally extend the singleton mechanism to provide more than one instance. We can store information for error localization within the `ErrorHandler`. Then, for example, we can replace special macros by adding functionality to the methods of the `ErrorHandler`.

Known Uses

Dart uses an `ErrorHandler` as shown in the sample code section. The exception handling concept developed for **HYPO-Bank** [GKL+95] contains an exception manager class which is comparable to the `ErrorHandler`. **EASY** implements exception handlers as functions. Their

error handling class allows dynamic configuration of exception handlers via a method `set_exception_handler`, which gets the function pointer as a parameter.

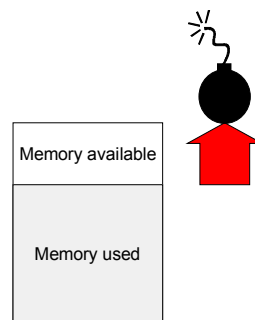
Resource Preallocation

Abstract

Resource Preallocation allows error handling to be operable even in troublesome out-of-memory situation.

Context

A program may use memory allocations for strings, objects, file handles, or other resources. If a program fails due to lack of these resources, we still want an operative error handling to get information describing the error situation. Here you have the perplexed situation that the error handling itself needs memory resources to work well.



Problem

How to ensure error processing although resources are short?

Forces

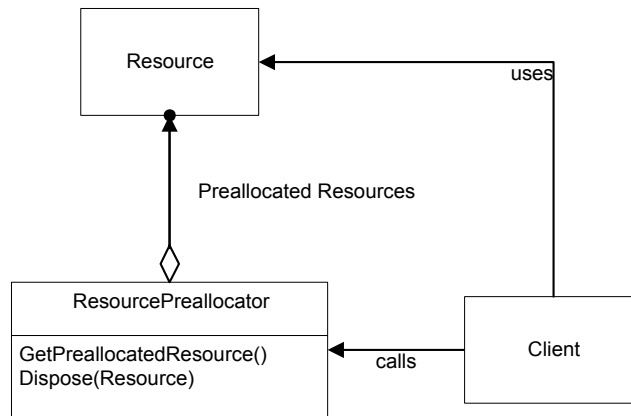
- Resource shortage is a frequent cause for exceptions.
- Separate resource management complicates error processing.
- Reserving resources decreased the resources available for standard operations and therefore increases the probabilities of resource shortage.
- Allocation of large chunks of memory in advance may result in unnecessary swapping.

Solution

Apply the following rules to design a failsafe error handling:

- Reduce dynamic memory allocation as much as possible.
- Preallocate files, dialogs and other necessary resources (e.g. a collection of preallocated error objects).
- Provide a memory „reserve tank“ for allocation that can be used when memory becomes short.

Structure



Participants

- **Resource**
 - any kind of object you want to preallocate for later use
- **ResourcePreallocator**
 - administrates a collection of preallocated resources
 - offers services to request and dispose the resources
- **Client**
 - needs resources managed by the ResourcePreallocator
 - uses the services of the ResourcePreallocator

Consequences

- This pattern allows the error handling to be operable even in troublesome out-of-memory situation. Hence, it is possible to shut down the system gracefully. In so far it may be indispensable for non-stop-systems or systems that are relevant to security. A system like a word processor may go down uncontrolled in case it runs out of memory. In any case controlled behavior is better to e.g. save the actual workspace.
- Memory leaks are permanent sources of memory loss. They can not be fought with this pattern. A code checker or heap checker like Purify or others will let you detect leaks easier than the point where your program runs out of memory by chance. An error backtrace can only show you the place where your program ran out of memory.
- It becomes possible to react to a lack of memory by freeing resources that might not be needed at the moment. So more options than simple shutdown of an application become available in case of a memory shortage.

- Memory preallocation uses more memory than necessary, as the amount of exceptions needed has to be allocated defensively. The maximum number of exceptions needed in a single threaded program is the maximum depth of the call stack plus a security factor.
- Memory preallocation is expensive in terms of programming cost and also a potential source of bugs. Compared to the operating systems or programming systems memory management facilities it is more likely that a custom memory manager might fail, so they should be stress tested.

Implementation

Besides a memory pool for error objects you should consider the following implementation issues when using preallocated resources for exception handling:

- In case you need a dialog box to prompt the user with an exception log, you may create this dialog box at program startup time and make it invisible. Most GUI libraries offer a `Hide()` method to make windows invisible without deallocating them.
- It may be necessary to have some memory for a controlled system shutdown like saving in a word processor. Don't forget to preallocate memory needed for such actions as well. The trick also works for other classes.
- Do not forget to preallocate log files and other resources for printing exception logs. Opening a file will usually cost a considerable amount of memory and might be impossible at the time all memory is used up.
- To redefine the memory management functionality several approaches are possible:
 - Implement an abstract Factory [GOF95] for all exception classes that also does memory management for exceptions.
 - Change the memory allocation algorithm to free pre-reserved memory.
 - Implement a special constructor with controlled memory management (e.g. by overriding the classes `new` and `delete` operators).

Sample Code

We now look at an implementation in C++. Due to the way exceptions are created in C++ by a `throw` statement, we cannot register an exception factory or use any other method to avoid the exception classes constructor. The exception classes normal constructor must be used. So what remains in C++ is a class specific `new` operator for exception classes that turns to a pool of preallocated exception objects.

We then have only one participant in the solution. The `ExBaseSafe` class is a variant of the above `ExBase` class that uses a static array of preallocated exception objects. The `new` and `delete` operators are replaced to allocate and deallocate new `ExBaseSafe` objects out of that static memory pool instead of allocating them with the `::new char[n]` function from the heap.

The following listing shows the class header (shortened version of `ExBase` above) plus the critical redefined `new` and `delete` operators:

```

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 & eExcep );

// 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 iPrototypes[NumberOfPreallocatedExceptions];

    // normal instance variables must be fixed length
    char pszExClass[ExClassIdLength];
        // unique identifier of an exception class
    ...
};

```

The new and delete operators are defined as follows:

```

void * ExBaseSafe::operator new(size_t size) {

    // a derived class forgot to define their own new and delete operators:
    EXC_ASSERT(size != sizeof( ExBaseSafe ));
    EXC_ASSERT(iFreePrototypeIndex > 0); // free exceptions left?

    return (&iPrototypes[iFreePrototypeIndex--]);

};

void ExBaseSafe::operator delete(void* deadObject, size_t size){

```

```
EXC_ASSERT(size != sizeof( ExBaseSafe );  
EXC_ASSERT(iFreePrototypeIndex < NumberOfPreallocatedExceptions -1 );  
  
iPrototypes[++iFreePrototypeIndex] = deadObject;  
  
};
```

Variants

Exception Class Hierarchy. In case you want a hierarchy of exception classes you might need to implement a new and delete operator for each one of them. This is similar to implementing a meta class for each class.

The **ANL** project implements resource preallocation in C++ using another trick, which is described in [Mey92b, item 8]. A new error handling function is defined using the C++ `set_new_handler`. This function may try to free memory somewhere in the system. The new operator may then retry to allocate memory or again invoke the custom error handling function.

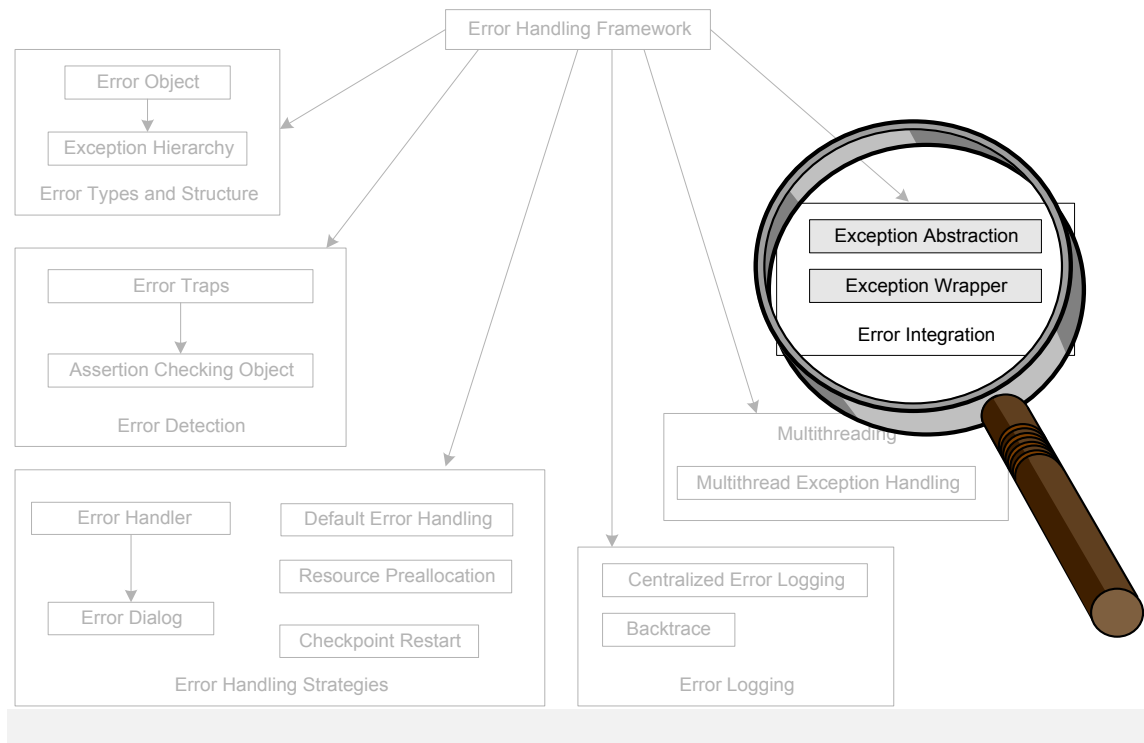
Known Uses

The error handling of the **CHAMPS** project preallocates a number of strings at system initialization.

Further Reading

For a deeper discussion of problems involved in memory management see [Mey92b] items 5 to 10. The idioms used to implement the above `ExBaseSafe` class are taken from [Mey92b].

2.6 Integration



Exception Abstraction

Context

In a layered architecture (Layers [BMR+96]) exceptions propagate across different layers: whenever a layer cannot handle an exception, it passes an exception to the next higher level. Because different layers present different levels of abstraction, each layer may only generate exceptions on its own abstraction level. Higher levels may have additional context information which allows more telling error messages.

Example

You are writing code that accesses a database. A query class is used to encapsulate queries to the database. The query issues SQL commands to the database and delivers results or throws exceptions in case the database operation failed for some reason.

For example, embedded or dynamic SQL offers countless, very detailed return codes that are delivered via the `sqlca.code` variable after command execution. As the caller of such an action you're not so much interested in why a database command failed.

It does not matter to the caller whether the database is down, a table is missing or an index is destroyed. The database does not deliver a result and that's a failure.

Problem

How do you generate reasonable error messages without violating abstraction levels?

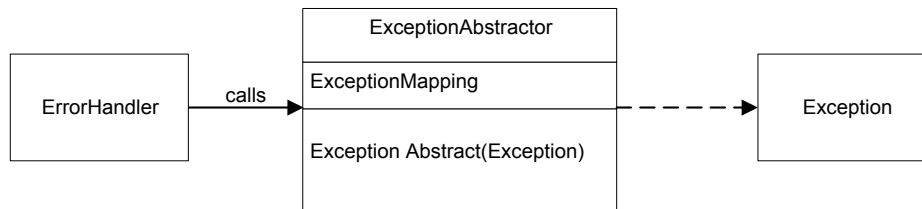
Forces

- Lower-level error messages may be useless and difficult to interpret on a higher level.
- Passing exceptions through different layers without abstraction will violate the principle of information hiding. It reduces reusability and clarity of the design
- Violating abstraction will result in an explosion of exception classes and error messages.
- The more layer boundaries an exception has to cross, the more it hinders performance.
- Providing the user of an interface with too much information results in higher programming cost.
- Providing not enough information also results in higher programming cost as programmers need workarounds and additional tricks to obtain the exception information needed.

Solution

Identify strategic parts of the system that recast exceptions according to the abstraction level rather than just forwarding them. Subsystem and layer boundaries are good candidates.

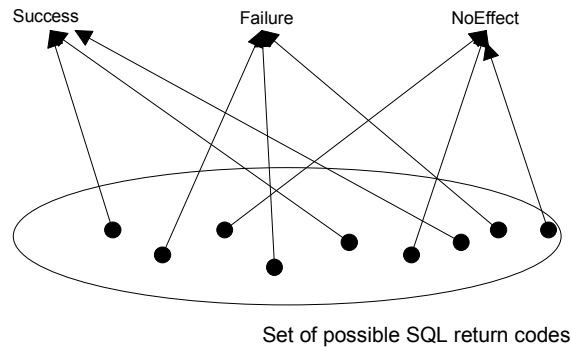
Structure



Example resolved

In the case of our example for a database access layer we may abstract the set of all possible results (e.g. signaled via return codes) to three different kind of results:

- SUCCESS the query did perform right. You will deliver a result.
- NO_EFFECT for cases where a query yielded no result or did not perform the action desired. You will deliver an exception and no result.
- SE for system errors like missing database links or such. You will deliver a severe exception. The user of your library may or may not shut down the system.



Consequences

- Exception Abstraction hides complexity from upper layers and delivers the information needed by upper layers.

Implementation

- You should consider logging the complete error information into a subsystems error log for later analysis by an administrator without bothering application programmers with information they cannot react on anyway. For example, the SQL code would be useful for a later analysis by a database administrator.
- If a subsystem or layer boundary contains several classes which perform the same abstraction mapping, you should implement a separate ExceptionAbstractor class to encapsulate the mapping. The error handlers of the interface classes uses the Exception-Abstractor to abstract lower-level exceptions and propagate them to the higher-level classes.

Known Uses

The example is taken from [Ebe90], where the set of all possible SQL codes is abstracted to a few return codes by a function called RC check_sql (sqlca.code).

The pattern may be found in most of sd&m's database access layers.

See Also

An Exception Wrapper may be a good place for Exception Abstraction.

Exception Wrapper

Abstract

Convert interfaces of classes that use an incompatible error handling concept. The new interface classes allow your code to use the functionality in a consistent way.

Context

You are integrating foreign libraries into your system. These might have some exception behavior that you do not like. Examples are:

- The libraries use a mixture of return codes and exceptions to prompt you with information on the success of operations.
- The libraries return NULL pointers instead of an exception if some operation fails.
- The libraries might throw unspecific or very fine grained exceptions that do not suit your needs.

Problem

How do you integrate a ready-to-use library into your exception handling system?

Forces

You will seldom find application frameworks that give you a common feel for programming. IBM's Collection Classes are totally different in their exception behavior from their Microsoft MFC counterparts. Booch Components are different from STL libraries and so on. As there is seldom any complete application frameworks today, you are forced to use various libraries from various vendors and integrate them into your programs.

This may result in unesthetic code or even in code that does not allow you to trace faulty situations. On the other hand, integrating different libraries from different vendors may be expensive in terms of programming effort for installing a common feel for programmers.

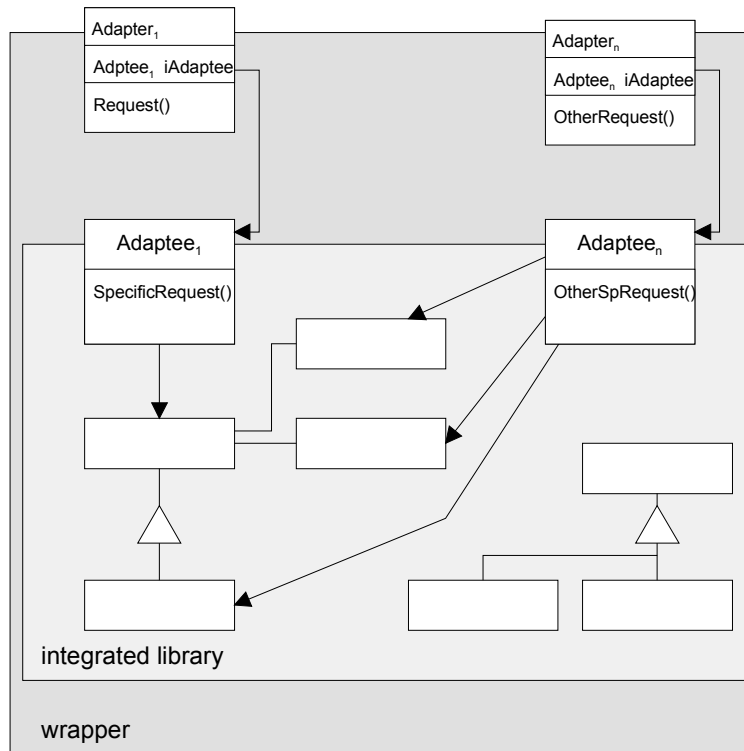
Solution

Use a wrapper class for each class that shows at the interface (Facade [GOF95]) of the library. Specify this wrapper class according to your exception handling scheme.

Structure

The figure below shows an example of wrapping. An integrated library consists of several classes. Some of these, the Adaptee Classes, form a Facade [GOF95]. The other unnamed classes need not be wrapped as they do not appear at the integrated cluster's client interface.

If you just want to adapt exception behavior, each Adaptee class is wrapped by an Adapter class according to the Adapter pattern [GOF95].



Consequences

Wrapping the exception behavior of a library will yield the following **advantages**:

- *Imitation of the Best*: Wrapping classes offers the opportunity to get high quality libraries by imitating the exception behavior and specifications of well designed libraries even if they are not available for your respective programming environment.
- *Common Feel for Programmers*: A programmer that uses several libraries from different vendors is given the illusion of a common programming style. This will result in less programming faults.
- *Improved Traceability of Exceptions*: You will not have to deal with unknown types of exceptions that are not properly integrated into your exception hierarchy when using foreign libraries somewhere below your code. You will instantly see which library prompted you with an exception and will be able to analyze a backtrace in any case.

The following **disadvantages** may be observed:

- *Programming Cost*: Wrapping a library means duplicating lots of interface code and having to specify large amounts of redundant interfaces.
- *Runtime penalty*: There is one extra level of calls caused by wrappers. Usually this will also result in extra object instantiations, which may turn out to be expensive.
- *Maintenance*: If the wrapped subsystem is updated (e.g. new version or bug fix) and the interfaces of the **Adaptee** classes change, you also have to update your **Adapter** classes.

Implementation

When implementing the above scheme you should think of the following implementation issues:

- Wrapping a „hacked“ class is a chance to clean its exception behavior using preconditions, postconditions and class invariants. Also think of improving the documentation.
- You need to implement exception classes according to your exception hierarchy.

Sample Code

The adaptation of exception behavior happens at method level. To demonstrate this aspect we need to look at an example in more detail. Lets presume we have a stack class that should be wrapped. The operation

```
Element* Stack::pop ( void )
{
    if isEmpty(iContainer) {
        return NULL;
    }
    else {
        return iContainer.firstElement();
    }
};
```

does not quite behave like we would like it to behave as it returns a NULL pointer in a case where we would like to see an exception.

We therefore wrap the stack with a class StackWrapper

```
class StackWrapper {
public:
    ....
    Element* pop( void );
    void push(Element* pushIt);
    ....
private:
    Stack    iStack;
};
```

and rewrite the exception behavior of Operation pop as follows:

```
Element* StackWrapper::pop(void) {
try {
    Element* pResult = NULL;
    pResult = iStack.pop();
    if (pResult == NULL) {
        _EX_THROW(ExStackWrapper, Ex_STW_EmptyStack, \
                „Trying to pop from Empty Stack“);
    }
    else {
        return pResult;
    }
};
}
```

```
catch (ExBase& anException) {
    // log and rethrows anything that is now converted
    // to our exception system
    ExceptionManager::Instance()->logException( anException );
    throw;
}
catch (...) {
    // we have an unexpected exception here that is not derived
    // from ExBase - that's a fatal software error
    _EX_THROW(ExBase,0,"Unexpected Exception");
}
};
```

The two catch blocks may well be packaged in a `_EX_WRAP_STD_CATCH` macro.

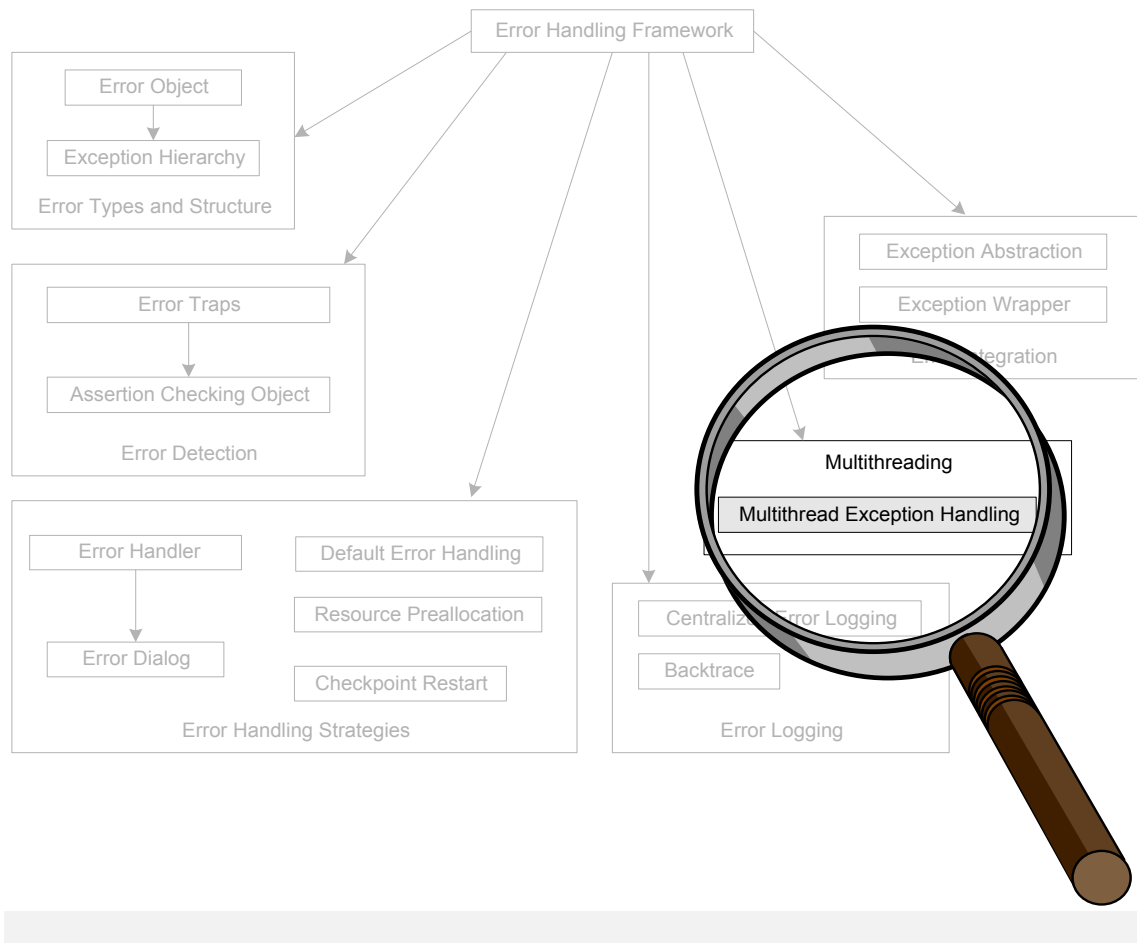
Known Uses

DATEV uses a wrapped version of the Common View GUI library that redefines exception behavior.

Further Reading

Wrapping is a standard technique often used in programming. See the [Adapter](#) pattern by [GOF95]. Instead of wrapping some libraries also offer the possibility to redefine macros that let you implement your own exception classes. See the IBM Container Class libraries.

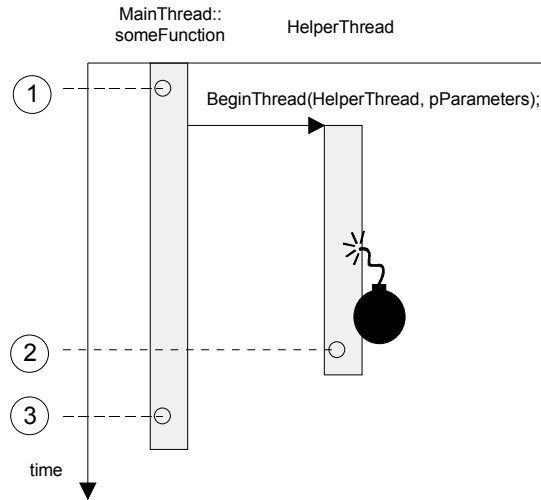
2.7 Multithreading



Multithread Exception Handling

Context

You are writing code for a multithreaded application. It is typical for such applications that a main thread starts another thread and delegates some tasks to it. The main thread will later try to collect the results.



Difficulties arise as parallel threads might be interrupted by exceptions. These exceptions must be signaled to and handled by the main thread.

Problem

How do you schedule exceptions in a multithread environment?

Forces

- The main driving force here is correctness. The error handling mechanism must be made thread safe. There should also be a log that allows exchange of exception information between threads.

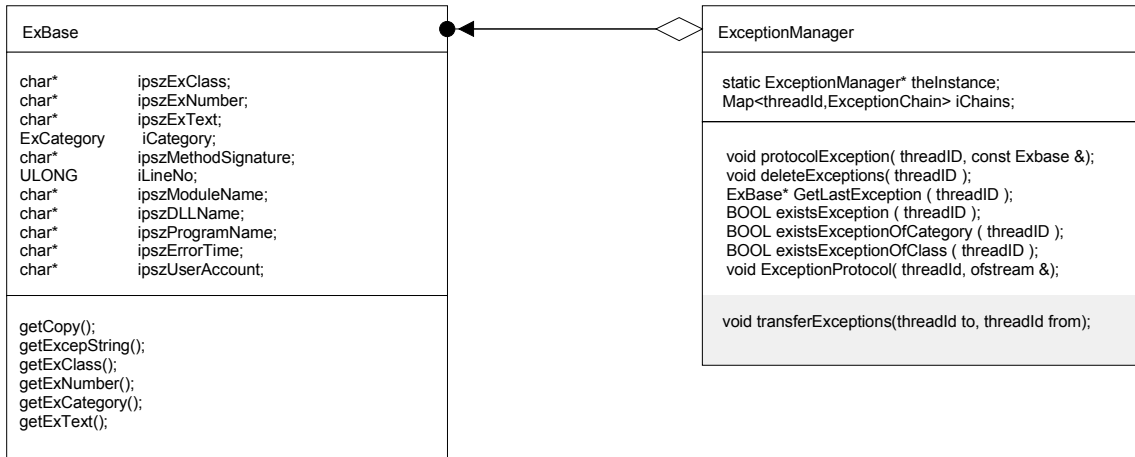
Solution

Use the above Error Handling Framework and extend it in a way that the ExceptionManager holds an own exception chain for each living thread.

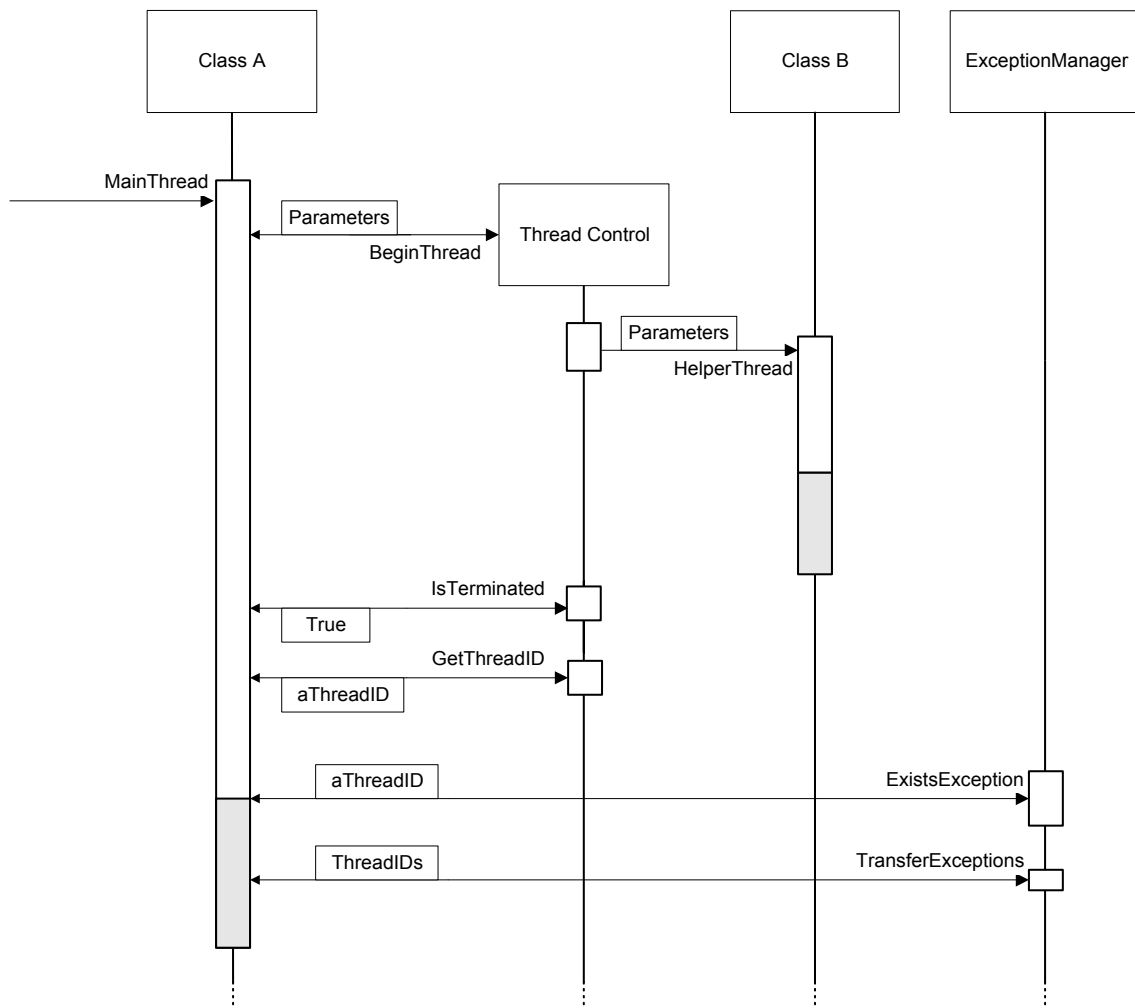
Structure

The structure is the same as in the Error Handling Framework pattern. The only difference is an additional parameter for threadIds in most of the ExceptionManager methods.

One additional method is required for the ExceptionManager. The transferExceptions method allows a surviving thread to transfer the dead threads exception chain into his own exception chain. The surviving thread is then able to rethrow the dead thread's last exception to the caller of the actual method he is processing.



Dynamics



Consequences

The solution described allows thread safe exception handling and exchange of exception information. Correctness is preserved by making the ExceptionManager thread safe. Exception

chains may be transferred so that the exception trace contains all information from the source of the error to the highest level of calls.

Implementation

When implementing the pattern, you have to take care of the following implementation issues:

- The Exception Manager must be made thread safe. This may be done by different means of the respective programming language. Some languages like JAVA or ADA offer language support for monitors. Some others like C or C++ require calling operating system's functions for semaphores or other policies of concurrency control.
- The protocolException() method needs an additional parameter indicating the thread id of the thread calling the ExceptionManager. This might be implemented using a ThreadId provided by the operating system or using a handle provided by the ExceptionManager.
- Programming environments are very different in the ways they handle multi thread programming. Therefore, have a look at your specific programming manual for how to implement the above examples. The above code is similar to 32bit Windows code but not exactly identical as nasty details like getting the threadId or obtaining termination information have been omitted for reasons of clarity of the example.

Sample Code

We will use a simple example to demonstrate the dynamic behavior of multithread exception handling.

```
UINT HelperThread( LPVOID pParam )
{
    CMyParamObject* pObject = (CMyParamObject*)pParam;

    // do something with 'pObject'
    try {
        doSomething( pObject );
        return 0;        // thread completed successfully
    }

    catch(ExBase & anException) { 2
        // some exception has occurred, derived from our
        // exception base class
        return 0;        // thread completed successfully with
        // respect to our programming model - we are still
        // in control of the situation
    }

    catch (...) {
        // we did catch an exception we cannot handle
        // perform a controlled abort of the program
        abort();
        return -1;
    }
};
```

```

// some function in the main thread that instantiates a new
// HelperThread

void MainThread::someFunction (void) {
try {

    // the new thread is started 1
    Thread* pHelperThread = NULL;
    pHelperThread = BeginThread(HelperThread, pParameters);
    // we will leave error checking for pHelperThread == NULL aside

    // the main thread does something else
    ...

    // some time later the main thread checks whether
    // the HelperThread did allright

    if (pHelperThread->isTerminated()) { 3
        if (ExceptionManager::_Instance()->
            existsException(pHelperThread->getThreadId()))
        {
            // the HelperThread terminated with an Exception
            // we could analyze it with getLastException()
            ExceptionManager::_Instance()->
                transferExceptions(this->getThreadId(),
                    pHelperThread->getThreadId());

            // we throw an own exception to leave our trace mark in
            // the exception chain
            _EX_THROW( ExBase, 0,
                "Unexpected Exception from HelperThread",
                ExBase::ExCat_Fatal );
        } // end if
    } // end if

    // continue normal operation of someFunction
} // end try
// the error is further propagated in the
// standard catch block
_EX_STD_CATCH

} // end someFunction

```

Known Uses

The above solution has been described by Henrik Ljungström for HYPO-Bank [GKL+95].

See also

This pattern gives only a short overview of the ideas, dynamics and of the classes involved. As [GKL+95] contains a detailed specification, we did not want to duplicate it here.

3

Design Rules

This chapter collects some general rules for designing the error handling part of business information systems. It should inform you about critical points. For example, if you have to look at an error handling design or if you have to implement such a design, it may be useful to keep these rules in mind.

Rule 1: Do not throw anything around. Use the above patterns!

If you are writing production code you should use the above patterns in a form adapted to your project's needs. You should therefore:

- Use an as simple as reasonable hierarchy of exceptions
- Use exception chains as proposed by the Error handling framework pattern
- Instrument each method or other piece of code with default error handling. Use macros or generators to keep the effort affordable.
- Log exceptions that you are not able to handle.

Rule 2: Do not mix a cocktail!

It is considered bad programming style to use a C (or Cobol)-Style return code concept in languages like C++ and Smalltalk. You may say: „*Where's the problem? Let me do it my way. Return codes may be so elegant*“.

But have you ever seen a Smart Pointer or an overloaded operator in C++ that yields a return code?

Return codes are primarily an idiom for 3GL languages. Be careful in applying this idiom in the context of object oriented languages. In 3GLs an operation contains much more code and control flow than a method in an object-oriented language. There are more if-cascades and result states which must be signaled to a caller. Another reason is the lack of support for handling exceptions. In contrast a modern object-oriented style can be characterized by

- Typeful programming.
- Many small objects with simple methods.
- Complexity within the interaction and dependencies between the small objects.
- Support for exception handling.
- Dynamic binding via inheritance.

Generally, it is no longer necessary to use return codes in these languages. It is better to use appropriate methods to ask for the state of a certain object or to introduce new object types. For exceptional behaviour, the exception mechanisms of the particular language should be used.

So if you program in an object-oriented language, avoid return codes. If you program in a classical procedural language return codes will be a good choice.

Rule 3: Be defensive - be kind to the people who have to do program maintenance

You should use preconditions, postconditions and class invariants for defensive programming. This allows easy tracking of programming bugs.

Rule 4: Do not break abstractions by using exceptions

It may be tempting to throw exceptions that contain all technical details of an error situation in a subcomponent. Be aware that this may break the abstraction of a subcomponent by giving the user detailed insight via the exception information.

It is better style to abstract exceptions at component borders and express the situation in terms of the component's contract that could not be fulfilled.

Rule 5: Do not use exceptions for the normal flow of control

The following is a NoNo:

```
// print aStack of Strings
try {
    while (TRUE) {
        char * pszPrintIt = aStack.pop();
        cout << aStack.pop() << endl
    }
}
catch(...) {
    // we come here if the stack is empty
}
// and so on ....
```



Rule 6: Cleanup your resources (especially important if there is no automatic garbage collector)

Cleanup your resources

- That were allocated and are not released automatically (files, objects, ...).
- That were allocated in constructors before an exception occurred. Who destroys objects not fully constructed because of an exception?
- Exception chains after they are handled or written to a log.

Check for tools on the market which support detection of memory errors and leaks (stack errors, heap errors).

Rule 7: Be careful with exceptions that may leave

Generally, exception propagation should be terminated within the following software units:

- Main functions (top-level control units).
- Start functions of threads.
- Event-handler functions in frameworks that do not support exception handling.
- Functions that may be called from C programs or other programs that cannot deal with exceptions.
- Destructors.

Your error handling architecture should contain design rules which describe what has to be done with exceptions in these particular units. What will happen to exceptions propagated by these units?

Rule 8: Subtype Conformance

If we use subtyping in an object-oriented specification we also have to consider subtype conformance. Besides the conditions concerning the arguments and results of methods (contravariance of arguments and covariance of results [LW94]) there is a third condition concerning exceptions:

<i>Covariance of exceptions:</i> If S is a subtype of T the exceptions for each method of S are a subset of those of T and the types of exceptions for each method of S are subtypes of those of T.

This rule ensures that a client of the supertype interface does not receive an exception declared in a subtype which is unknown to him.

In the Java programming language, one has to specify the exceptions thrown by a method and the compiler checks whether the methods fulfill the covariance rule and detects unspecified exceptions.

Rule 9: Separation of concerns

Often, there is no clear separation of different kind of exceptions (either there is no separation at all or the criteria are fuzzy and unclear). For example, only one class is defined for all kind of return states. As a result it becomes nearly impossible to distinguish between error handling as a reaction to unexpected behaviour and exception handling as a part of the specified behaviour. With a fuzzy, unstructured concept, there is no chance to get a consistent and reliable implementation and much time is spent on discussions and explanations.

Rule 10: Keep control

Imagine every developer is himself responsible for handling exceptions adequately. Everyone individually decides whether to signal another exception, to recover or to mask the exception without reaching a consistent state. A new exception is defined within a special file. In the beginning the file is very small and nobody cares. Then the file becomes larger and larger and finally there is a chaotic collection of exceptions with a lot of redundancy (rising entropy). Thus it is important to document the design rules, e.g. which layer is responsible for which exceptions and what is the overall strategy for error handling within each layer? But it is not sufficient to write these constraints down in some design document. These rules must be enforced too (against time, budget), either by inspections and reviews or, even better, by tools.

Rule 11: Consistency

The error messages are often stored in a file together with corresponding message identifiers. Each message string can contain special placeholders for parameters which are replaced by actual values at run-time. During development there is the problem of consistency between the message file and the statements written by the developers in the source code. Handling everything as strings and implementing placeholders as numbered tokens will probably result

in type errors, wrong sequences of parameters and wrong message identifiers. It is also very uncomfortable for the programmer to manually ensure consistency.

So try to use tools to automate the handling of error messages or use features of your programming language to get more checks done by the compiler.

Acknowledgments

Special thanks to my ARCUS team colleagues Jens Coldewey and Wolfgang Keller. Wolfgang carefully reviewed the first draft of this paper, when it was just a single, bulky pattern, and pointed out the way to a pattern language. He also contributed with experiences from the Hypo project and compiler-tested C++ sample code. Jens continuously reviewed earlier versions; I went into depression each time I saw his endless number of comments and suggestions for improvement. I also thank Fridtjof Toenniessen (the official sd&m-„shepherd“ for this paper) for his comments. Last but not least, I am grateful to Falk Carl and Gerhard Albers who gave me insight to the error handling of TLR and DaRT. Their input greatly influenced the pattern language, especially the source code was very useful.

Appendix A

Glossary

Backward Error Recovery

Tries to overcome an error by setting back to an earlier consistent state and possibly retry afterwards. In contrast to forward error recovery, no detailed analysis of the error condition is necessary.

Error

Part of system state that is liable to lead to \Rightarrow *failure*. Manifestation of a \Rightarrow *fault* in a system.

Error code

Encoding for the type of a \Rightarrow *fault*. It gives a classification of different error situations.

Because no \Rightarrow *exception mechanism* is available in 3-GLs, error propagation must be done by normal control and data flow. Therefore, error codes are widely used (as special return codes or as an additional output parameter) to signal error information to the caller of an operation.

Exception

Any occurrence of an abnormal condition during the execution of a software unit that causes an interruption in normal control flow is an *exception*. An *exception is raised* when such a condition is signaled by a software unit. In response to an exception the control is immediately given to a designated handler for the exception, which reacts to that situation (*exception handler*). The handler can try to recover (▣▣▣ *fault tolerance*) from that exception in order to continue at a predefined location or it cleans up the environment and further escalates the exception.

Often the term exception is also (mis-)used as a synonym for ▣▣▣ *error*, abnormal response, and ▣▣▣ *failure*.

Exception Mechanism

An exception mechanism is a language control structure allowing programmers to describe the replacement of the standard program execution by an exceptional execution when an occurrence of an ▣▣▣ *exception* is detected. This mechanism is usually an essential part of any modern language.

Exceptions can be handled according to different models: The *termination model* requires a handler to leave the enclosing execution block after completion. The *resumption model* allows the handler to recover the program state and continue execution from the operation following the causing one. But an exception can also be propagated when recovery is not feasible.

Most languages implement the termination model (e.g. C++, Ada) because it is more practical and reliable than the more complex resumption model.

Fault

A judged or hypothesized cause of an ▣▣▣ *error*. One can distinguish between design faults, hardware faults, lower level service faults, and specification faults.

Failure

Deviation of the delivered service from compliance with the specification. Transition from correct service delivery to incorrect service.

Fault-Tolerance

Fault-tolerant software detects ▣▣▣ *errors* caused by ▣▣▣ *faults* and employs error recovery techniques to restore normal computation. Depending on the kind of fault there exists different techniques to achieve fault-tolerance. For instance, we can distinguish between design fault-tolerance, hardware fault-tolerance and lower level service fault-tolerance.

Concerning techniques for error recovery, there are two general approaches which are known as ▣▣▣ *forward* and ▣▣▣ *backward error recovery*.

Forward Error Recovery

Tries to reach or reconstruct a correct state out of the current erroneous state by special repair actions, often by means of redundancy and analysis of the detected error.

System Safety

The ability of a system to prevent catastrophic failures.

Reliability

The ability of a system to deliver its normal service in presence of \Rightarrow errors and unexpected or incorrect usage (user errors). Two aspects of reliability can be distinguished: \Rightarrow *Fault Tolerance* and \Rightarrow *Robustness*.

Return code

Is a return value which encodes information about the success or \Rightarrow *failure* of an operation. It is used by the caller to distinguish between a set of possible result states.

If a return value encodes error information in case of failure it is also called an \Rightarrow *error code*. Error codes are very common in 3-GLs.

Robustness

A robust program must always terminate in a defined way, so that behaviour is predictable for all possible inputs. These include protection against incorrect usage, degenerate inputs and all kinds of errors.

Appendix B

Checklist

During the design process, we have to answer a number of questions. The solution or design space is populated by all possible answers to these questions.

Our pattern language presents a set of patterns which cover a part of the design space and contains successful and common solutions. Each pattern provides an answer to one specific design question. We now give a list of such questions. It should help the reader to consider important design aspects and it is also a good index for the pattern language because it gives the references to the particular patterns which are helpful to answer these questions.

Detection (Runtime Checks)

What to check? Integrity, Parameters, Preconditions, Postconditions, Types?	⇒ <i>Error Traps (42)</i>
When to check? At the interface of each method, watch-dog processes?	⇒ <i>Error Traps (42)</i>
How to check? Special methods?	⇒ <i>Assertion Checking Object (50)</i>
Runtime configuration of checks?	⇒ <i>Assertion Checking Object (50)</i>

Error Protocol

Different kind of protocols? Who is the reader of the protocol?	
Content?	⇒ <i>Backtrace (55)</i>
When to write the data?	
Who has control?	

Configurability?

Errors

How do you prevent an unstructured bulk of exceptions, defined by different developers without any coordination? ⇒ *Exception Hierarchy (37)*

How do you master the complexity of a large number of exceptions?

What type of exceptions exists? An object type for each exception? ⇒ *Error Object (31)*

Classification of exceptions? Type hierarchy? What characteristics are important and must be distinguished? What are the criteria? ⇒ *Exception Hierarchy (37)*

How to implement exception types? ⇒ *Error Object (31)*

What is the context of each exception? Who can raise a particular exception and where are handlers for this exception? What is the handling strategy for that exception? Is it an exception internal to a component? How far can the exception be propagated? Can an exception cross different layers? Where are these rules and constraints defined and who controls them?
⇒ *Error Handler (80)*
⇒ *Default Error Handling (65)*
⇒ *Exception Abstraction (90)*

Error View

Are there different views for errors? Default View? ⇒ *Error Dialog (68)*

Changeability of user messages?

Where are user message texts stored? In the code, in a file or in a database? ⇒ *Error Object (31)*

Error Handler

What kind of error handlers are implemented? ⇒ *Default Error Handling (65)*
⇒ *Checkpoint Restart (76)*

Tool Support

What kind of information can be generated? Code frames? Complete Classes?

Specification

What kind of errors must be specified? ⇒ *Appendix Appendix C*

How do you specify errors? ⇒ *Appendix Appendix C*

What about subtyping and errors?

⇒ *Rule 8 (105)*

Who performs type checking?

Consistency

Is there a repository or database which contains all information mentioned in the model? Can a developer lookup information in the database?

Are consistency checks automated or done by hand?

What kind of consistency is checked?

Appendix C

Some Theory

This appendix tries to elucidate the context of error handling to get a better grip on the different aspects of exception handling.

For a more thorough and formal treatment of exception handling in the context of fault tolerance you may look at [Chr94].

C.1 Error Specification

It becomes clear in Chapter 1.2 that failure can only be defined in relation to a specification. It is nonsense to talk about failure without any specification, as it is nonsense to talk about correctness without a specification. We also mentioned that the design of reliable software components cannot be done as an afterthought, it strongly affects the design of the component's interfaces. So the main work has to be done during the specification of the components. In every method the boundary between normal and exceptional behaviour must be defined.

If we take a more abstract view on the specification of a method, we can think of a method as a state-transition function. The domain of such a function is given by all the input-states from a state-space which yield a defined result. The state-space may be defined by the attributes of the class the method belongs to and recursively of all reachable classes (e.g. by the method parameters or other references). Whether the function is total or partial depends on the precondition. If there is no precondition at all (always TRUE) the state space is not constrained and the domain comprises the whole state space (see Figure 14). The more constraints are added in form of preconditions, the smaller the domain becomes within the state space. If such a method is called with an input-state out of the domain, the behaviour is unspecified.

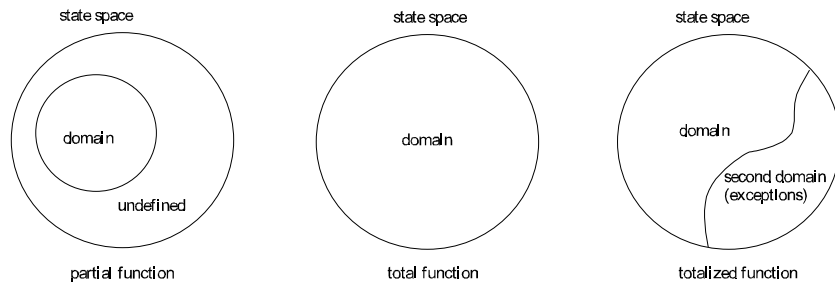


Figure 14: domain of a function

Exceptions permit us to structure and extend an operation's domain (or its range). With exceptions we can tailor a method's results to the particular purpose in using the method. In this respect exceptions are used to totalize methods (see Figure 14), so that their behaviour is also defined when preconditions are violated or other exceptional situations occur.

If we look at the actual (run-time) behaviour of a function, we can partition the input state space in four disjunctive sets :

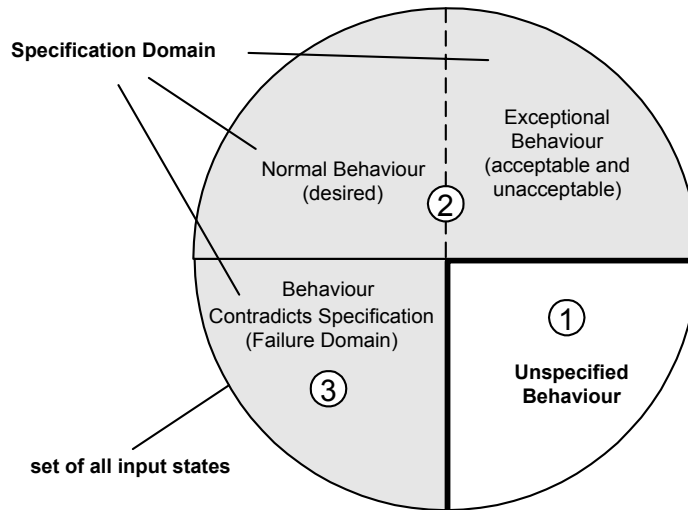


Figure 15: Run-time view of behaviour

There is one partition ❶ of unspecified behaviour defined by the preconditions for that function (partial function). A second partition ❷ comprises all states for which the behaviour of the function corresponds to the specified behaviour. This partition is subdivided into normal and exceptional behaviour (which does not mean error). The last partition is the failure domain ❸. It contains valid input states (according to the function's preconditions), but the function's behaviour deviates from the specified one.

The goal during specification, design and implementation is to get an empty failure domain as well as no unspecified behaviour. How do we reach this goal?

In the specification of a function we partition the state space into three domains:

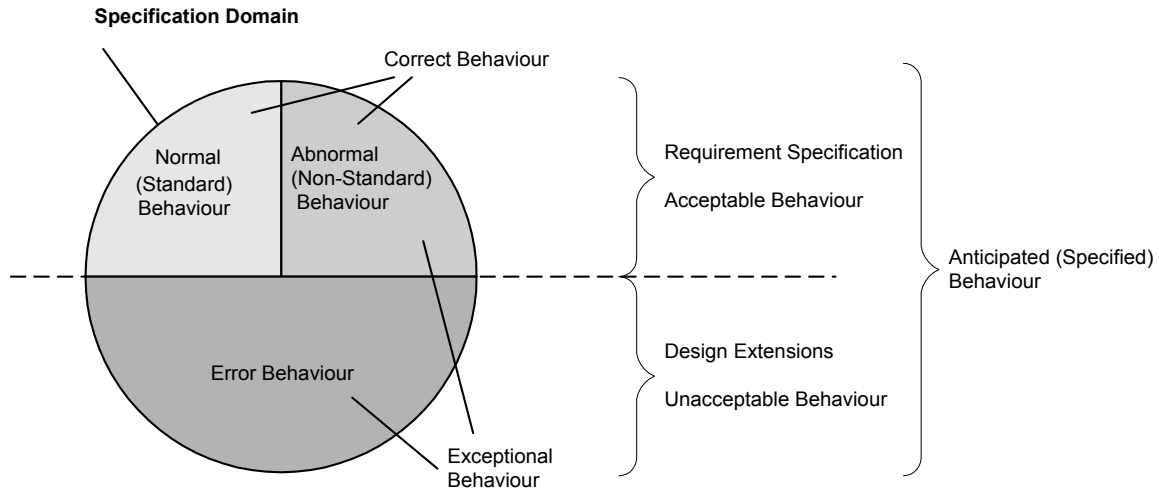


Figure 16: Specification view of behaviour – the three domain approach

The most important one (especially in the early phases) covers the core functionality. We call this core functionality the normal (or standard) behaviour of a function. Additionally, we have to consider a lot of more special cases which are not very usual but they must be handled properly — it's the abnormal (or non-standard) behaviour. Together, standard and non-standard behaviour define the correct⁶ behaviour. The third domain is that of errors which is defined as part of the technical design. This domain tries to cover all other states and defines some reasonable behaviour for them. No general recipe exists where to draw the line between non-standard, standard, and error behaviour (otherwise good design could be automated). Therefore, it is the main source of lengthy discussions and uncertainties.

Examples:

The cash limit of an ATM from Chapter 1.2.2 is a good example for correct, abnormal behaviour, which must be specified in the interface of the corresponding method. We assume that an account is modeled by a class and some methods exist to check conditions about an account:

```

CLASS Account

DATA
  AccountNr      :      accnr;
  Person         :      owner;
  PIN            :      ownerid;
  Balance        :      balance;

METHODS
  Bool VerifyOwner(Person);
  Bool VerifyPIN(PIN);
  Bool CheckLimit(Amount);

```

⁶ It is difficult to find a good term for the correct exceptional behaviour. If the word normal is used in the sense of correct it can also be used for the correct exceptional behaviour.

```
Bool IsBlocked();
...
```

The withdraw transaction is another class with a method `Execute`:

```
CLASS Withdraw
-- A transaction class.
-- Instances of Withdraw are produced by the ATM with the data typed in
-- by a user. Processing is done on the bank side.
DATA
    AccountNr      :      card_accnr;
    Amount         :      req_amount;
    PIN            :      card_pin;

METHOD Execute
--
-- A withdraw-transaction executed on the bank side.
--

INPUT      -- no explicit parameters, but card_accnr, req_amount, card_pin
              -- are implicit parameters.

EXCEPTIONS
    ExBlockedAccount; ExOverLimit; ExIncorrectPIN; ExAccountNotExist;

REQUIRES
    card_accnr, req_amount and card_pin must be defined.

NORMAL BEHAVIOUR
    The database is looked up for the account specified by card_accnr.
    The balance for this account is reduced by req_amount.

EXCEPTIONAL BEHAVIOUR
    An account card_accnr is not found in the database
        => RAISE ExAccountNotExist;
    The account card_accnr is blocked => RAISE ExBlockedAccount;
    req_amount exceeds the limit => RAISE ExOverLimit;
    The verification of card_pin fails => RAISE ExIncorrectPIN;
```

By invocation of `Execute` it is assumed that the ATM has already checked two things: First, the entered PIN corresponds to the PIN on the card. The second assumption is that the cash dispenser has enough money for the transaction. These conditions are essential for the use-case and must be specified on a higher level. Thus `Execute` models only the bank side part of the use case. A withdraw transaction requires an account number, a pin and the requested amount of money (specified as a precondition introduced by the keyword `REQUIRES`). E.g. the method's behaviour is unspecified if we call it without an assigned account number. Such an account number would be part of the domain ❶ in Figure 15. But in the implementation of this method the unspecified domain is covered by the error handling which checks the precondition and reacts in a suitable way (see Figure 16).

Consider a more advanced ATM which allows the user a lot of other transactions. The ATM as a client gets a handle on the `Account` and can verify the PIN directly at the beginning of a

session, before starting any other transactions. Now the verification of the PIN is not part of the `Execute` method's behaviour anymore:

```
METHOD Execute

INPUT      -- no explicit parameters, but req_amount and a handle to the
              -- user's account are implicit parameters.

EXCEPTIONS
    ExOverLimit;

REQUIRES
    account and req_amount are defined. Account is not blocked.
...

```

The method operates on a given account. Authentication of the user and verification whether the account is blocked must be done before calling `Execute`.

Normally, it would be better to minimize the exceptional cases and to separate different tasks. But for the specification of methods no universal panacea exists. Sometimes it would be more elegant to constrain a method by preconditions and in other situations exceptional behaviour can be used to get more unspecific and reusable operations. Another possibility (foremost in the design of reusable classes) is to provide different versions of a method within an interface (e.g. by overloading). One version could throw exceptions for abnormal conditions whereas the other could not. So the user can decide which version he may use in his context.

Consider now, that we call `Execute` with correct parameter values and the result would be an exception `ExOverLimit`, although the actual amount does not exceed the limit. This would be a failure, because the method does not behave according to the specification (the input values are part of the failure domain ☹). □

Generally, the correct behaviour is specified with the assumption of perfect hardware and software. It comprises all functional behaviour (including all non-standard cases) relevant to the application.

Error Behaviour is concerned with all kinds of component failure due to an imperfect world (lack of resources, incorrect implementation, hardware defects, ...).

We can further divide the error domain into two subdomains:

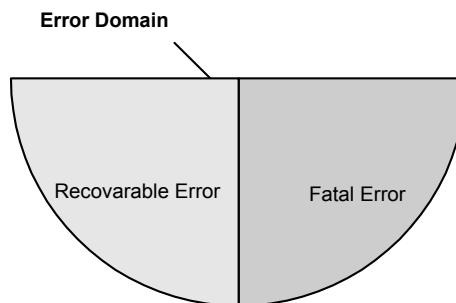


Figure 17: Further partitioning of the error specification domain

Fatal Errors are those states which do not allow any repair actions, the application must be terminated. Recoverable errors are less severe situations which may be handled by the system.



- Be aware of the border between errors and abnormal conditions which is partly a matter of specification.
- An abnormal condition in a lower level component may cause an error in a higher level component.
- Error handling is concerned with all aspects of error domain specific design and implementation.
- Exception handling comprises handling of abnormal situations as well as errors.

You can think of error handling as a separate run-time system (or as an extension of an existing run-time system). This run-time system works like an undercover agent performing run-time inspections of the program state and raising exceptions in case of suspicious situations. The fatal errors detected by it can occur everywhere in the system (so that there are not specified for each method) and there is a defined place within the system where the errors are handled. Much of the code for these run-time checks can be generated automatically from a specification (if the specification is precise enough).

It is most important to clearly separate these different domains and to implement the specification in a consistent and coherent way – this is the key for successful error handling!

The following table summarizes some criteria to distinguish errors from correct abnormal and normal behaviour:

(System) Error Handling	Handling Abnormal Conditions	Normal Behaviour
<ul style="list-style-type: none"> • produces error messages • must be very robust and fault-tolerant • special system error protocol (backtracing the call-stack) • inherent (technical) part of every method, a client need not know about it when using a method 	<ul style="list-style-type: none"> • may result in (user) error messages and warnings • (user) error protocol (no back-trace) • exceptions are specified as part of the interface, a client must expect these exceptions 	<ul style="list-style-type: none"> • may result in general messages, help messages and other informational messages • configurable trace protocol • central part of the specification (desired behaviour)

Table 1: Errors vs. Abnormal Conditions vs. Normal Behaviour



- Note that we use three different protocols:
1. *System Error Protocol*: Used by system administrators and system developers to analyze serious error conditions and to fix possible bugs.

2. *User Error Protocol*: Used by system administrators and system users (especially during batch processing) to analyze abnormal behaviour as a result of user errors (wrong access rights, missing data, incorrect usage,...).
3. *Trace Protocol*: Used by system developers merely during the test phase but it can also be useful during maintenance (especially if the error protocol is not sufficient to find a bug).

C.2 Some remarks about implementation

During implementation a number of new questions arise: What mechanisms of the programming language should be used for which domain? Do exception handling mechanisms only apply to erroneous situations or also to abnormal situations? How do you separate the domains within the programming language? This chapter should give some answers to these questions.

Modern programming languages support the handling of exceptions by special mechanisms and keywords which help to separate the normal code and control-flow from that of exception handling (e.g. Modula-3 [Har92], Eiffel [Mey88], Ada, C++, Java, Smalltalk).

In programming languages which are not object-oriented and also do not support exception handling (e.g. 3GL-languages like C and Cobol), *return codes* are often used instead. They code information about the state of a routine. Information about a failure may be available via a global variable or an additional parameter.

The classical return-code mechanism is also called a nice-guy approach, because the caller of a method is responsible for paying attention to upcoming error information. One advantage of exceptions is that they can't be totally ignored by the caller. If somebody raises an exception, control is automatically given to an exception handler. In most cases the return-code is a number which represents the status of a method. Don't use the normal return path for system errors when possible. Exceptions should be used as an emergency exit. If there is a fire in a building, you do not use the normal exit either. An exception is a concept which is independent from a concrete implementation language. But of course it is easier to implement an exception if this concept is directly supported by the programming language. Otherwise, the exception mechanism has to be implemented or „simulated“ by yourself with the existing „traditional“ programming features. This leads to a mass of code for checking return values and propagating error values. Using exceptions makes the code cleaner and easier to read.

Table 2 gives an overview of different mechanisms and their preferable usage.

	Normal Behaviour	Abnormal Behaviour	Error Behaviour
Exception Mechanism	☠ Never!	O +	+
Return Value (Code)	+	O	-
Output Parameter	+	O +	-

Global Data	–	O	O
--------------------	---	---	---

Table 2: Design choices – When to use which mechanism?

Of course, there might be a number of reasons why a certain mechanism might be used or not. In case of error handling the use of exceptions is beyond dispute, but for abnormal behaviour it is more controversial. For example [Mey88, p. 147] warns of the misuse of exceptions, in Modula-3 [Har92] even the normal return and exit statements are implemented by exceptions, Java uses exceptions very extensively for abnormal behaviour as well as for error handling.⁷

Table 3 presents the most favorable choices for some programming languages.

	Normal Behaviour	Abnormal Behaviour	Error Behaviour
Java, C++, Smalltalk	Parameter, Return Value	Exceptions	Exceptions
Cobol, Cobol	Parameter, Return Value	Return Codes	Global Data/Return Codes

Table 3: Design choices for different programming languages

To conclude, an exception handling is merely a structural tool within modern programming languages which can be very helpful for error handling. An implementation of error handling is obviously more straightforward and less laborious in languages with a built-in exception handling concept than in other languages.

C.3 Idealized model for error handling

In a „design by contract“ approach, each method is specified by a contract in form of pre- and post-conditions. On the one hand, this contract specifies the normal and abnormal behaviour, and, on the other hand, it outlines the border to the error domain. What happens if the precondition is violated or other methods which are called could not fulfill their contracts for whatever reason must be defined during design. In such a case, an error exception is raised.

According to Meyer [Mey88] we want to distinguish the following error situations for every method *m*:

- | |
|--|
| <ol style="list-style-type: none"> 1. The pre- or postcondition of <i>m</i> is found to be violated. 2. A class invariant is found to be violated on entry or termination. 3. An assertion violation (e.g. invalid parameter, loop invariant not maintained by a loop iteration, variant not decreased, ...) is found during the execution of <i>m</i>. |
|--|

⁷ The concept and support for exception handling seems to be much better in Java than in C++, but today there are no experiences with Java as a programming language for large information systems. If at all Java is primarily used for the user interface part of an information system (e.g. in combination with CORBA).

- | |
|--|
| <ol style="list-style-type: none">4. A method called by m fails.5. An operation executed by m results in an error condition detected by the hardware or the operating system. |
|--|

These situations can be divided into two categories:

- *external errors*: this group comprises all exceptions which are raised by the environment of a method (context), e.g. hardware, operating system, neighboured systems, other used methods of that class
- *internal errors*: these are all kind of assertion violations which may occur within that method

If an error occurs there are three possible responses:

- *Retry*: the method tries to recover from the error situation and executes the failed routine again.
- *Organized Panic*: the method is unable to solve the problem and therefore only tries to cleanup the state and then signals the error to the caller (which can also mean to give a message to the user).
- *False Alarm*: because of additional information the method recognizes that the error is not really an error in that context and neutralizes the error.

In the second case, the caller of the method which failed has to respond according to one of these three choices again.⁸

These considerations are summarized in the following diagram:

⁸ For a system user it is mostly the same: in organized panic he will inform the system administrator (the human counterpart of an error-handler) or calls the support which is also a kind of error signaling.

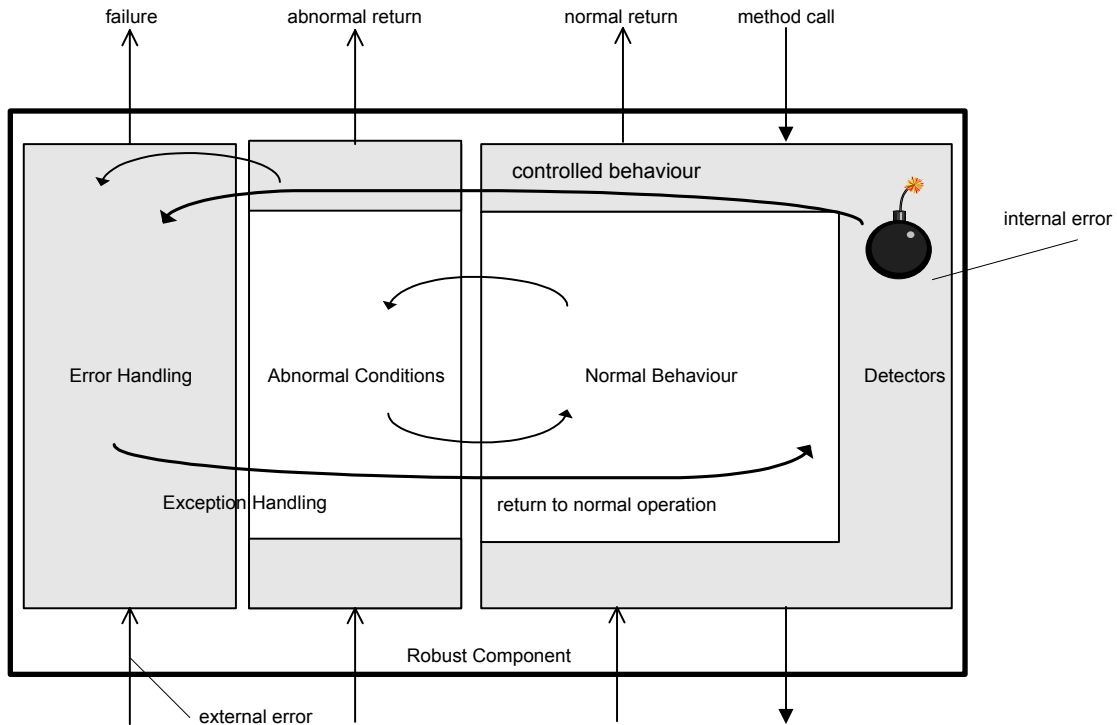


Figure 18: Model of a robust component

In this model a component (class, module, system or even a single operation) is divided into three parts according to the three specification domains we discussed so far. The „normal part“ is guarded by detectors: On the one hand, they check all kinds of input (e.g. parameters of method calls and return values of lower level methods invoked by the component) and on the other hand they perform internal checks (e.g. consistency of the component state, component invariants). If an error is found by a detector (illustrated by the bomb symbol in Figure 18), the error handling part of the component is activated (on the left in Figure 18). Either the error handling is able to recover from that error (shown by the arrow labeled „return to normal operation“) or it signals a failure to the caller of the method. The error handling also becomes active by external errors (a failure signaled to the component as a result of another method called by that component).

Example: If we look at this component as a model for a whole system, a method call might correspond to a service which is requested by a user. If the system cannot fulfill this service and must signal a failure to the user because of an error we call this a *system error*. Of course, not every subsystem (or subcomponent) failure must result in a system failure (remember the ATM example). If the user requests a service with incorrect parameters we expect the user error handling within the user interface component to recover from this error and to guide the user until correct parameters are given.

Appendix D

The Ariane 5 Failure

(Excerpt from the report of the Inquiry Board)

„On 4 June 1996, the maiden flight of the Ariane 5 launcher ended in a failure. Only about 40 seconds after initiation of the flight sequence, at an altitude of about 3700 m, the launcher veered off its flight path, broke up and exploded. Engineers from the Ariane 5 project teams of CNES and Industry immediately started to investigate the failure.

...

2. ANALYSIS OF THE FAILURE

2.1 CHAIN OF TECHNICAL EVENTS

In general terms, the Flight Control System of the Ariane 5 is of a standard design. The attitude of the launcher and its movements in space are measured by an Inertial Reference System (SRI). It has its own internal computer, in which angles and velocities are calculated on the basis of information from a *strap-down* inertial platform, with laser gyros and accelerometers. The data from the SRI are transmitted through the databus to the On-Board Computer (OBC), which executes the flight program and controls the nozzles of the solid boosters and the Vulcain cryogenic engine, via servovalves and hydraulic actuators.

In order to improve reliability there is considerable redundancy at equipment level. There are two SRIs operating in parallel, with identical hardware and software. One SRI is active and one is in *hot* stand-by, and if the OBC detects that the active SRI has failed it immediately switches to the other one, provided that this unit is functioning properly. Likewise there are two OBCs, and a number of other units in the Flight Control System are also duplicated.

The design of the Ariane 5 SRI is practically the same as that of an SRI which is presently used on Ariane 4, particularly as regards the software.

Based on the extensive documentation and data on the Ariane 501 failure made available to the Board, the following chain of events, their inter-relations and causes have been established, starting with the destruction of the launcher and tracing back in time towards the primary cause.

- The launcher started to disintegrate at about H0 + 39 seconds because of high aerodynamic loads due to an angle of attack of more than 20 degrees that led to separation of the boosters from the main stage, in turn triggering the self-destruct system of the launcher.
- This angle of attack was caused by full nozzle deflections of the solid boosters and the Vulcain main engine.
- These nozzle deflections were commanded by the On-Board Computer (OBC) software on the basis of data transmitted by the active Inertial Reference System (SRI 2). Part of these data at that time did not contain proper flight data, but showed a diagnostic bit pattern of the computer of the SRI 2, which was interpreted as flight data.
- The reason why the active SRI 2 did not send correct attitude data was that the unit had declared a failure due to a software exception.
- The OBC could not switch to the back-up SRI 1 because that unit had already ceased to function during the previous data cycle (72 milliseconds period) for the same reason as SRI 2.
- The internal SRI software exception was caused during execution of a data conversion from 64-bit floating point to 16-bit signed integer value. The floating point number which was converted had a value greater than what could be represented by a 16-bit signed integer. This resulted in an Operand Error. The data conversion instructions (in Ada code) were not protected from causing an Operand Error, although other conversions of comparable variables in the same place in the code were protected.
- The error occurred in a part of the software that only performs alignment of the strap-down inertial platform. This software module computes meaningful results only before lift-off. As soon as the launcher lifts off, this function serves no purpose.
- The alignment function is operative for 50 seconds after starting of the Flight Mode of the SRIs which occurs at H0 - 3 seconds for Ariane 5. Consequently, when lift-off occurs, the function continues for approx. 40 seconds of flight. This time sequence is based on a requirement of Ariane 4 and is not required for Ariane 5.
- The Operand Error occurred due to an unexpected high value of an internal alignment function result called BH, Horizontal Bias, related to the horizontal velocity sensed by the platform. This value is calculated as an indicator for alignment precision over time.
- The value of BH was much higher than expected because the early part of the trajectory of Ariane 5 differs from that of Ariane 4 and results in considerably higher horizontal velocity values.

The SRI internal events that led to the failure have been reproduced by simulation calculations. Furthermore, both SRIs were recovered during the Board's investigation and the failure context was precisely determined from memory readouts. In addition, the Board has examined the software code which was shown to be consistent with the failure scenario. The results of these examinations are documented in the Technical Report.

Therefore, it is established beyond reasonable doubt that the chain of events set out above reflects the technical causes of the failure of Ariane 501.

2.2 COMMENTS ON THE FAILURE SCENARIO

In the failure scenario, the primary technical causes are the Operand Error when converting the horizontal bias variable BH, and the lack of protection of this conversion which caused the SRI computer to stop.

It has been stated to the Board that not all the conversions were protected because a maximum workload target of 80% had been set for the SRI computer. To determine the vulnerability of unprotected code, an analysis was performed on every operation which could give rise to an exception, including an Operand Error. In particular, the conversion of floating point values to integers was analysed and operations involving seven variables were at risk of leading to an Operand Error. This led to protection being added to four of the variables, evidence of which appears in the Ada code. However, three of the variables were left unprotected. No reference to justification of this decision was found directly in the source code. Given the large amount of documentation associated with any industrial application, the assumption, although agreed, was essentially obscured, though not deliberately, from any external review.

The reason for the three remaining variables, including the one denoting horizontal bias, being unprotected was that further reasoning indicated that they were either physically limited or that there was a large margin of safety, a reasoning which in the case of the variable BH turned out to be faulty. It is important to note that the decision to protect certain variables but not others was taken jointly by project partners at several contractual levels.

There is no evidence that any trajectory data were used to analyse the behaviour of the unprotected variables, and it is even more important to note that it was jointly agreed not to include the Ariane 5 trajectory data in the SRI requirements and specification.

Although the source of the Operand Error has been identified, this in itself did not cause the mission to fail. The specification of the exception-handling mechanism also contributed to the failure. In the event of any kind of exception, the system specification stated that: the failure should be indicated on the databus, the failure context should be stored in an EEPROM memory (which was recovered and read out for Ariane 501), and finally, the SRI processor should be shut down.

It was the decision to cease the processor operation which finally proved fatal. Restart is not feasible since attitude is too difficult to re-calculate after a processor shutdown; therefore the Inertial Reference System becomes useless. The reason behind this drastic action lies in the culture within the Ariane programme of only addressing random hardware failures. From this point of view exception - or error - handling mechanisms are designed for a random hardware failure which can quite rationally be handled by a backup system.

Although the failure was due to a systematic software design error, mechanisms can be introduced to mitigate this type of problem. For example the computers within the SRIs could have continued to provide their best estimates of the required attitude information. There is reason for concern that a software exception should be allowed, or even required, to cause a processor to halt while handling mission-critical equipment. Indeed, the loss of a proper software function is hazardous because the same software runs in both SRI units. In the case of Ariane 501, this resulted in the switch-off of two still healthy critical units of equipment.

The original requirement accounting for the continued operation of the alignment software after lift-off was brought forward more than 10 years ago for the earlier models of Ariane, in order to cope with the rather unlikely event of a hold in the count-down e.g. between - 9 seconds, when flight mode starts in the SRI of Ariane 4, and - 5 seconds when certain events are initiated in the launcher which take several hours to reset.

The period selected for this continued alignment operation, 50 seconds after the start of flight mode, was based on the time needed for the ground equipment to resume full control of the launcher in the event of a hold.

This special feature made it possible with the earlier versions of Ariane, to restart the count-down without waiting for normal alignment, which takes 45 minutes or more, so that a short launch window could still be used. In fact, this feature was used once, in 1989 on Flight 33.

The same requirement does not apply to Ariane 5, which has a different preparation sequence and it was maintained for commonality reasons, presumably based on the view that, unless proven necessary, it was not wise to make changes in software which worked well on Ariane 4.

...

Returning to the software error, the Board wishes to point out that software is an expression of a highly detailed design and does not fail in the same sense as a mechanical system. Furthermore software is flexible and expressive and thus encourages highly demanding requirements, which in turn lead to complex implementations which are difficult to assess.

An underlying theme in the development of Ariane 5 is the bias towards the mitigation of random failure. The supplier of the SRI was only following the specification given to it, which stipulated that in the event of any detected exception the processor was to be stopped. The exception which occurred was not due to random failure but a design error. The exception was detected, but inappropriately handled because the view had been taken that software should be considered correct until it is shown to be at fault. The Board has reason to believe that this view is also accepted in other areas of Ariane 5 software design. The Board is in favour of the opposite view, that software should be assumed to be faulty until applying the currently accepted best practice methods can demonstrate that it is correct.

This means that critical software - in the sense that failure of the software puts the mission at risk - must be identified at a very detailed level, that exceptional behaviour must be confined, and that a reasonable back-up policy must take software failures into account. ...“

References

By and large the literature on practical and reusable error handling solutions is very sparse. Most publications are either concerned with special problems of exception handling in programming languages (above all C++) [Car95, EC95a, EC95b, Har92, Lea96, Mey92b, Mey96, Mül96, Ree96, Spu94, Tal96] or discuss error handling from a more general (e.g. fault tolerant systems, system safety) and theoretic view [Chr94, Fed90, Goo75, Lev86, Lee90]. The pattern community has not given much attention to this topic yet either.

The following remarks should give some orientation which reference might be interesting to look up for further information.

[Den91] is a good introduction to the principles of error handling. He mentions the main things which have to be considered and sketches a „classical“ and well-tried solution for handling errors in business information systems. It does not cover exception handling mechanisms which can be found in modern programming languages.

For further information about specific problems concerning exception handling in C++ a lot of good papers and books are available [Car95, Ell95a, Ell95b, Lea96, Mey92, Mey96, Mül96, Ree96]. Most of it is published in the C++ Report. The general message of this contributions is: exception handling in C++ is not such an easy job as it may look like. There are some side-effects when you introduce throw/catch-blocks and exceptions.

[Ell95a] introduces the notion of *exception safety*: a class cannot become inconsistent due to an exception thrown during the execution of a member function. This is especially important if you want to recover from an error condition and proceed with normal execution. Their work shows how difficult it is to develop exception safe software in C++.

Scott Meyers [Mey92, Mey96 pp. 44-80] gives a very useful list of idioms for exception handling in C++. Both books are highly recommended for every C++ project at sd&m.

A book which is still a good reference also for this topic is Meyer's standard book on object-oriented software development [Mey88a]. He discusses and motivates the handling of abnor-

mal conditions fairly well and explains the connection between the contract-metaphor and error handling.

A general, more academic introduction to exception handling is given in [Goo75]. In contrast to the other papers it also discusses the broad context in which exception handling may be used.

A formal framework for exception handling mechanisms in object-oriented programming languages can be found in [Fed90].

- [Ada96] **D. Adams:** *Mostly Harmless*. Book 5 of the Hitch Hiker's Guide to the Galaxy trilogy, Heinemann, London, 1992.
- [BMR+96] **Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal:** *Pattern Oriented Software Architecture, A System of Patterns*, Wiley 1996.
- [Car95] **Tom Cargill:** *Exception handling: A false sense of security*. C++ Report, Volume 6, No. 9, Nov.-Dec. 1995.
- [CSF96] **Castek Software Factory:** *Component-Based Development: Exception Handling Specifications*. [<http://www.castek.com>], White Paper, April 1996.
- [Chr94] **F. Christian:** *Exception Handling and Tolerance of Software Faults*. In: M. R. Lyu (ed.): *Software Fault Tolerance*, J. Wiley, 1995, 81-107.
- [Den91] **E. Denert:** *Software-Engineering*, Springer Verlag, 1991.
- [Ebe90] **B. Ebersberger:** *Richtlinien und Werkzeuge zur Erstellung portabler Datenbank-Anwendungssoftware auf UNIX unter besonderer Berücksichtigung von Makrotechnik*. Master Thesis, Technische Universität München, 1990.
- [EC95a] **Margaret Ellis, Martin Carroll:** *Designing and Coding Reusable C++*. Addison Wesley, 1995.
- [EC95b] **Margaret Ellis, Martin Carroll:** *Tradeoffs of Exceptions*. C++ Report, Volume 7, No. 3, March-April 1995.
- [Fed90] **C. Feder:** *Ausnahmebehandlung in objektorientierten Programmiersprachen*. Informatik-Fachberichte, No. 235, Springer Verlag, 1990.
- [Goo75] **John Goodenough:** *Exception Handling: Issues and a Proposed Notation*. Communications of the ACM, December 1975.
- [GKL95] **K. Grüb, W. Keller, H. Ljungström:** *Konzept für die Fehlerbehandlung in C++ für die HYPO-Bank*. sd&m, 1995.
- [GOF95] **Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides:** *Design Patterns, Elements of Reusable Object-oriented Software*, Addison-Wesley 1995.
- [Har92] **S. P. Harbison:** *MODULA-3*. Prentice Hall, 1992
- [KW95] **W. Keller, K. Wagner:** *Fehlerbehandlung in großen C++ Projekten*. sd&m, 1995.

- [Lap92] **J. C. Laprie (ed.):** *Dependability: Basic Concepts and Terminology, Dependable Computing and Fault Tolerance*, Springer-Verlag, 1992.
- [Lea96] **S. Leary:** *C++ Exception Handling in Multithreaded Programs*. C++ Report, Volume 8, No. 2, February 1996.
- [LA90] **P. A. Lee, T. Anderson:** *Fault Tolerance: Principles and Practice*. Springer-Verlag, 1990.
- [Lev86] **N. G. Leveson:** *Software Safety: Why, What, and How*. ACM Computing Surveys, 18, No. 2, June, 1986.
- [LW94] **B. Liskov, J. M. Wing:** *A behavioral notion of subtyping*. ACM TOPLAS, 16(6), November 1994, 1811-1841.
- [Mey92a] **B. Meyer:** *Object-oriented Software Construction*. Prentice Hall, 1988.
- [Mey92b] **Scott Meyers:** *Effective C++*. Addison Wesley 1992.
- [Mey96] **Scott Meyers:** *More Effective C++*. Addison Wesley 1996.
- [Mül96] **H. M. Müller:** *Ten Rules for Handling Exception Handling Successfully*. C++ Report, Volume 8, No. 1, January 1996.
- [Neu95] **P. G. Neumann:** *Computer-Related Risks*. Addison Wesley, 1995.
- [Ree96] **J. W. Reeves:** *Exceptions and Standard C++*. C++ Report, Volume 8, No. 5, May 1996.
- [RBP+96] **J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen:** *Object-Oriented Modelling and Design*. Prentice-Hall, 1991.
- [Spu94] **D. A. Spuler:** *C++ and C Debugging, Testing, and Reliability*. Prentice Hall, 1994.
- [Sur96] **H. Surrer:** *Batch-Design*. sd&m, internal lecture notes, August 1996.
- [Tal96] **Taligent:** *Taligent's Guide To Designing Programs*. Addison Wesley, 1994.
- [TLR95] **TLR/IHK:** *Entwicklerhandbuch*. Version 4.0, sd&m, 1995.