# More Patterns for the Generation, Handling and Management of Errors

*Andy Longshaw and Eoin Woods*

**Abstract**

As systems become more complex it is increasingly difficult to anticipate and handle error conditions in a system. The developers of the system must ensure that errors do not cause problems for the users of the system. In a previous paper [Longshaw 2004] a collection of patterns for such distributed error handling was explored. As this collection was refined, two new patterns emerged: **Hide Technical Details from Users** and **Unique Error Identifier**. This paper retains the same context and but is focused on obtaining feedback specifically on these new additions.

## Introduction

In recent years there has been a wider recognition that there are many different stakeholders for a software project. Traditionally, most emphasis has been given to the end user community and their needs and requirements. Somewhere further down the list is the business sponsor; and trailing well down the list are the people who are tasked with deploying, managing, maintaining and evolving the system. This is a shame, since unsuccessful deployment or an unmaintainable system will result in ultimate failure just as certainly as if the system did not meet the functional requirements of the users.

One of the key requirements for any group required to maintain a system is the ability to detect errors when they occur and to obtain sufficient information to diagnose and fix the underlying problems from which those errors spring. If incorrect or inappropriate error information is generated from a system it becomes difficult to maintain. Too much error information is just as much of a problem as too little. Although most modern development environments are well provisioned with mechanisms to indicate and log the occurrence of errors (such as exceptions and logging APIs), such tools must be used with consistency and discipline in order to build a maintainable application. Inconsistent error handling can lead to many problems in a system such as duplicated code, overly-complex algorithms, error logs that are too large to be useful, the absence of error logs and confusion over the meaning of errors. The incorrect handling of errors can also spill over to reduce the usability of the system as unhandled errors presented to the end user can cause confusion and will give the system a reputation for being faulty or unreliable. All of these problems are manifest in software systems targeted at a single machine. For distributed systems, these issues are magnified.

In a previous paper [Longshaw 2004], we set out a collection (or possibly a language) of patterns that relate to the use of error generating, handling and logging mechanisms – particularly in distributed systems. The patterns in this collection are not about the creation of an error handling mechanism such as [Harrison] or a set of language specific idioms such as [Haase] but rather in the application code that makes use of such underlying functionality. The intention is that these patterns combine to provide a landscape in which sensible and consistent decisions can be made about when to raise errors, what types of error to raise, how to approach error handling and when and where to log errors.

This paper extends and refines this original pattern collection by defining two new, related patterns, namely Unique Error Identifier and Hide Technical Details from Users.

## Overview

The patterns presented in this paper form part of a pattern collection aimed at guiding the designers of error handling in multi-tier distributed information systems. Such systems present a variety of challenges with respect to error handling, including the distribution of elements across nodes, the use of different technology platforms in different tiers, a wide variety of possible error conditions and an end-user community that must be shielded from the technical details of errors that are not related to their use of the system. In this context, a software designer must make some key decisions about how errors are generated, handled and managed in their system. The patterns in this collection are intended to help with these system-wide decisions. This type of far-reaching design decision needs careful thought and the intent of the patterns is to assist in making such decisions.

As mentioned above, the patterns presented here are not detailed design solutions for an error handling framework, but rather, are a set of design principles that a software designer can use to help to ensure that their error handling approach is coherent and consistent across their system. This approach to pattern definition means that the principles should be applicable to a wide variety of information systems, irrespective of their implementation technology. We are convinced of the applicability of these patterns in their defined domain. You may also find that they are applicable to systems in other domains - if so then please let us know.

The patterns that comprise the entire collection are illustrated in Figure 1. The boxes in the diagram each represent a pattern in the collection. The arrows indicate dependencies between the patterns, with the arrow running from a pattern to another pattern that it is dependent upon.

The patterns in the grey boxes are the ones covered in this paper. All of the other patterns are described in thumbnail form in the appendix. The full form of the other patterns is available in the paper submitted to EuroPLoP 2004 [Longshaw 2004].

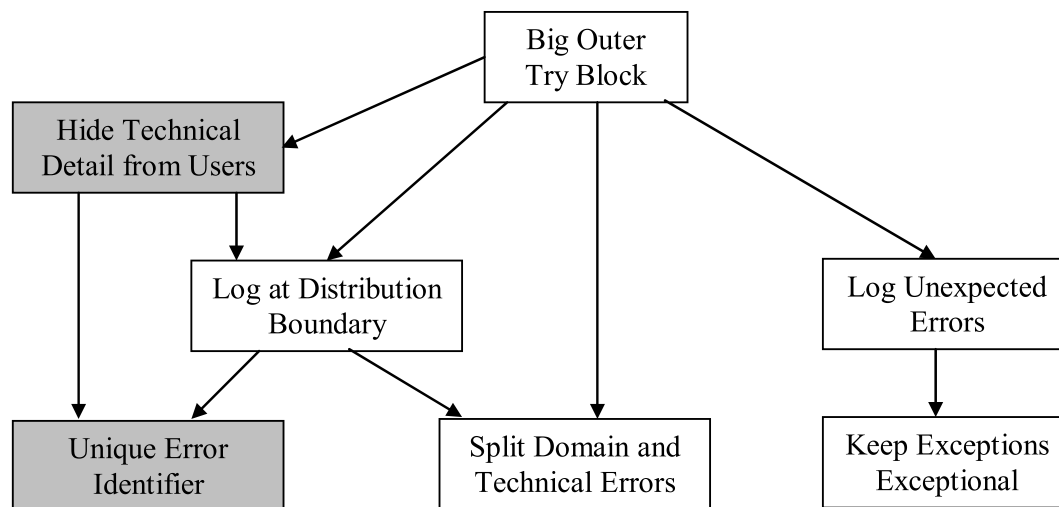


**Figure 1 - Error Handling Patterns**

The relationships between the patterns are as follows:

- *Log Unexpected Errors* depends upon *Make Exceptions Exceptional* so that expected conditions do not become exceptions and get incorrectly logged.
- *Log at Distribution Boundary* depends upon *Split Domain and Technical Errors* so that the two broad error categories can be handled differently.
- *Log at Distribution Boundary* depends upon *Unique Error Identifier* to mitigate the potential confusion arising from one error causing multiple log entries.
- *Hide Technical Details from Users* depends upon *Log at Distribution Boundary* so that the errors that it receives are suitable to use as a basis for display to the user
- *Hide Technical Details from Users* also depends (directly or indirectly, according to use) upon *Unique Error Identifier* to mitigate the potential confusion arising from one error causing multiple log entries.
- *Big Outer Try Block* depends upon *Split Domain and Technical Errors* so that the two broad error categories can be handled differently
- *Big Outer Try Block* depends upon *Log at Distribution Boundary* so that the errors that it receives are more relevant and potentially suitable for display to the user.
- *Big Outer Try Block* depends upon *Hide Technical Detail from Users* so that appropriate messages are displayed to users.

The two highlighted patterns are described in the main body of the paper. The remaining patterns, plus several proto-patterns, are briefly described at the end of the paper.

# Unique Error Identifier

## Problem

If an error on one tier in a distributed system causes knock-on errors on other tiers you get a distorted view of the number of errors in the system and their origin.

## Context

Multi-tier information systems, particularly those that use load balancing at different tiers to improve availability and scalability. Within such an environment you have already decided that as part of your error handling strategy you want to *Log at Distribution Boundary*.

## Forces

- It is often possible to determine the sequence of knock-on errors across a distributed system just by correlating raw error information and timestamps but this takes a lot of skill in system forensics and usually a lot of time.
- The ability to route calls from a host on one tier to one of a set of load-balanced servers in another tier improves the availability and scalability characteristics but makes it very difficult to trace the path of a particular cross-tier call through the system.
- You can correlate error messages based on their timestamp but this relies on all server times being synchronized and does not help when two errors occur on servers in the same tier within a small time window (basically the time to make a distributed call between tiers).
- Similar timestamps help to associate errors on different tiers but if many errors occur in a short period it becomes far harder to definitively associate an original error with its knock-on errors.

## Solution

Generate a Unique Identifier for each error that occurs and propagate this back to the caller. Always include the Unique Identifier with any error log information so that multiple log entries from the same cause can be associated and the underlying error can be correctly identified.

## Known Uses

The authors have observed this pattern in use within a number of successful enterprise systems. We do not know of any publicly accessible implementations of it (because most systems available for public inspection are single tier systems and so this pattern is not relevant to them).

## Implementation

The two key tenets that underlie this pattern are the uniqueness of the error identifier and the consistency with which it is used in the logs. If either of these are implemented incorrectly then the desired consequences will not result.

The unique error identifier must be unique across all the hosts in the system so that a particular error event can be reliably identified. This rules out many pseudo-unique identifiers such as those guaranteed to be unique within a particular virtual platform instance (.NET Application Domain or Java Virtual Machine). The obvious solution is to use a platform-generated Universally Unique ID (UUID) or Globally Unique ID (GUID). As these utilize the unique network card number as part of the identifier then this guarantees uniqueness in space (across servers). The only issue is then uniqueness across time (if two errors occur very close in time) but the narrowness of the window (100ns) and the random seed used as part of the UUID/GUID should prevent such problems arising in most scenarios.

It is important to maintain the integrity of the identifier as it is passed between hosts. Problems may arise when passing a 128-bit value between systems and ensuring that the byte order is correctly interpreted. If you suspect that any such problems may arise then you should pass the identifier as a string to guarantee consistent representation.

The mechanism for passing the error identifier will depend on the transport between the systems. In an RPC system, you may pass it as a return value or an [out] parameter whereas in SOAP calls you could pass it back in the SOAP fault part of the response message.

In terms of ensuring that the unique identifier is included whenever an error is logged, the responsibility lies with the developers of the software used. If you do not control all of the software in your system you may need to provide appropriate error handling through a *Decorator* [Gamma 1995] or as part of a *Broker* [Buschmann 1996]. If you control the error framework you may be able to propagate the error identifier internally in a *Context Object* [Fowler].

## *Consequences*

The positive consequences of using this pattern are as follows.

- The system administrators can use a unified view of the errors in the system keyed on the unique error identifier to determine which error is the underlying error and which other errors are knock-ons from this one. If the errors in each tier are logged on different hosts it may be necessary to retrieve and amalgamate multiple logs in a *System Overview* [Dyson 2004] before such correlation can take place.
- Correlating errors based on the unique error id rather than the hosts on which they occur gives a far clearer picture of error cause and effect across one or more tiers of load-balanced servers.
- Skewed system times on different servers can cause problems with error tracing. If an error occurs when host 1 calls host 2, host 2 will log the error and host 1 will log the failed call. If the system time on host 1 is ahead of host 2 by a few milliseconds, it could appear that the error on host 1 occurred before that on host 2 – hence obscuring the sequence of cause and effect. However, if they both have the same unique error identifier, the two errors are inextricably linked and so the time skew could be identified and allowed for in the forensic examination.
- If lots of errors are generated on the same set of hosts at around the same time it becomes possible to determine if a consistent pattern or patterns of error cascade is occurring.

The negative consequences of using this pattern are as follows.

- The derivation of a unique error identifier may be relatively complex in some environments and this could be a barrier to the pattern's adoption in some situations.
- The implementation of this pattern implies logging each error a number of times, once in each tier. This additional logging activity means that overall, logs will grow more quickly than in systems that do not implement this approach. This means that the runtime and administration overhead of this additional logging will need to be absorbed in the design of the system.

## Related Patterns

- *Log at Distribution Boundary* needs errors to have a unique error id in order to correlate the distributed errors.
- You may or may not employ *Centralized Error Logging* [Renzel 97] to help assimilate errors.

# Hide Technical Error Detail from Users

## *Problem*

The technical details of errors that occur are typically of no interest to the end-users of a system. If exposed to such users, this error information may cause unnecessary concern and support overhead.

## *Context*

An information systems application, with a largely non-technical user community, probably using the system via some sort of graphical interface.

## *Forces*

- If a detailed error report, particularly for a technical error, is presented to an end user, they are likely to find its content incomprehensible.
- If technical errors are presented to end users or the application simply stops or crashes unexpectedly then this is likely to cause a loss of confidence in the application, possibly leading to a reluctance to use it.
- Inconsistent user error reporting makes the system difficult to support as it confuses the users and prevents them reporting problems accurately and consistently.
- Technical errors generally have a lot of information that is useful for Support Staff but it is irrelevant to the end user.
- If the system under consideration offers a limited capability user interface (such as that offered by a mobile device), the interface may not be capable of reporting detailed error information in a comprehensible manner.

## *Solution*

Implement a standard mechanism for reporting unexpected technical errors to end-users. The mechanism can report all errors in a consistent way at a level of detail appropriate to the different user constituencies who need to be informed about the error.

## *Known Uses*

The authors are aware of a number of instances of this pattern in enterprise systems, although none of them are available for public study. Some examples of using this pattern outside the domain of enterprise systems include the following.

- A number of *self service web-sites* report a generic error message if an internal error occurs, including a unique error identifier that can be used to report the situation to a helpdesk.
- Some *intelligent hardware devices* respond to errors that occur by displaying a simple error screen (in some cases including a unique error identifier to allow the error to be uniquely identified by the hardware supplier), that instructs the owner to call a telephone hotline in order to obtain assistance.
- The *Microsoft Windows error dialog* that is displayed when an application encounters an internal error is an example of the use of this pattern.

## Implementation

Within the system's user interface implementation, provide a single, straightforward mechanism for reporting technical errors to end-users. The mechanism is almost certainly going to be a simple API call of the general form:

```
void notifyTechnicalError(Throwable t) ;
```

The mechanism created should perform two key tasks:

- Log the full technical details of the error that has occurred for possible use by technical support staff.
- Display a friendly, user-centric message to inform the user that something terrible has happened in general terms, making it clear that what has happened is not related to their use of the system. The user message should include some form of unique identifier along with a clear instruction to guide the user to report what has happened and the error identifier (if necessary), via some form of helpdesk.

  Ideally, the user reporting of the error should be automated in some way (for example using desktop email automation) in order to make the process of reporting as simple as possible and to avoid errors during the process. If the process is automated, this will avoid the problem of users ignoring the errors because reporting them is too much trouble and will ensure accurate reporting of each error.

  From the information in the user's error report, a helpdesk can escalate the problem to an administrator who can access detailed error information elsewhere in the system, using the identifier as a key.

Use this mechanism to handle all technical errors encountered by the system's user interface.

## Consequences

Positive consequences of implementing this pattern are as follows.

- Users of the system are never presented with technical error information that could confuse or worry them.
- The system becomes easier to support because support staff can correlate fatal system errors with logged information in order to allow them to understand and investigate the problem.
- Error handling in the GUI implementation is simplified and standardized.

Negative consequences of implementing this pattern are as follows.

- Concealing all error information from the end-user means that a knowledgeable end-user is powerless to apply their own knowledge to solve the problem. This could mean that a number of avoidable calls are made to helpdesks, that could otherwise be resolved by the users themselves.
- The implementation of this pattern may require the implementation of a reasonably sophisticated error-handling framework and this may be perceived as a significant overhead within the development process.

### *Related Patterns*

- This pattern fits very naturally with the *Big Outer Try Block* to ensure that technical errors are displayed and logged appropriately.
- Using the *Log at Distribution Boundary* pattern to govern where technical errors are logged ensures that the received are suitable for reporting to the end user and include a suitable unique identifier.
- This pattern can alternatively be combined directly with *Unique Error Identifier* to ensure that errors can be clearly identified.
- An *Error Dialog* [Renzel 97] forms part of a strategy to hide errors from users.

# Existing Pattern Reference

## *Split Domain and Technical Errors*

### Problem

Applications have to deal with a variety of errors during execution. Some of these errors, that we term “domain errors”, are due to errors in the business logic or business processing (e.g. wrong type of customer for insurance policy). Other errors, that we term “technical errors”, are caused by problems in the underlying platform (e.g. could not connect to database) or by unexpected faults (e.g. divide by zero). These different types of error occur in many parts of the system for a variety of reasons. Most technical errors are, by their very nature, difficult to predict, yet if a technical error could possibly occur during a method call then the calling code must handle it in some way.

Handling technical errors in domain code makes this code more obscure and difficult to maintain.

### Solution

Split domain and technical error handling. Create separate exception/error hierarchies and handle at different points and in different ways as appropriate.

## *Log at Distribution Boundary*

### Problem

The details of technical errors rarely make sense outside a particular, specialized, environment where specialists with appropriate knowledge can address them. Propagating technical errors between system tiers results in error details ending up in locations (such as end-user PCs) where they are difficult to access and in a context far removed from that of the original error.

### Solution

When technical errors occur, log them on the system where they occur passing a simpler generic SystemError back to the caller for reporting at the end-user interface. The generic error lets calling code know that there has been a problem so that they can handle it but reduces the amount of system-specific information that needs to be passed back through the distribution boundary.

## *Big Outer Try Block*

### Problem

Unexpected errors can occur in any system, no matter how well it is tested. Such truly exceptional conditions are rarely anticipated in the design of the system and so are unlikely to be handled by the system’s error handling strategy. This means that these errors will propagate right to the edge of the system and will appear to “crash” the application if not handled at that point. This may lead to some or all of the information associated with such unexpected errors being lost, leading to difficulties with the rectification of underlying problem in the system.

### Solution

Implement a Big Outer Try Block at the “edge” of the system to catch and handle errors that cannot be handled by other tiers of the system. The error handling in the block can report errors in a consistent way at a level of detail appropriate to the user constituency.

## *Log Unexpected Errors*

### Problem

Much domain code includes handling of exceptional conditions and is designed to recognize and handle each condition according to a business process definition (typically the offending transaction being rejected or a new domain entity being created). If such routine error conditions are logged, this makes real errors requiring operator intervention difficult to spot.

### Solution

Implement separate error handling mechanisms for expected and unexpected errors. Error conditions that are expected to arise in the course of normal domain processing should not be logged but handled in the code or by the user. Hence, any logged error should be viewed as requiring investigation.

## *Make Exceptions Exceptional*

### Problem

A number of languages include exception handling facilities and these are powerful additions to the error handling toolkit available to programmers. However, if exceptions are used to indicate expected error conditions occurring, then calling code becomes much more difficult to understand.

### Solution

Indicate expected domain errors by means of return codes. Only use exceptions to indicate runtime problems such as underlying platform errors or configuration/data errors.

# Proto-Patterns

## *Ignore Irrelevant Errors*

### Problem

Sometimes technical errors or exceptions do not denote a real problem and so reporting them can just be confusing or irritating for support staff.

### Solution

Assess what action can be taken in response to an error and only log it if there is a relevant course of action. Example is ThreadAbortException which is raised under ASP.NET whenever you transfer to another page using Server.Transfer(). This is not an error condition – just a side-effect – and so is of no consequence to support staff. Also, you will get lots of these in any busy web-based system.

## *Single Type for Technical Errors*

### Problem

There are a myriad different technical errors that may occur during a call to an underlying component.

### Solution

When you create your exception/error hierarchy for your application, define a single error type to indicate a technical error, e.g. SystemError. The definition and use of a single technical error type simplifies interfaces and prevents calling code needing to understand all of the things that can possibly go wrong in the underlying infrastructure. This is especially useful in environments that use checked exceptions (e.g. Java).

## References


| | |
|---|---|
| Buschmann 96 | *Pattern-Oriented Software Architecture*, John Wiley and Sons, 1996 |
| Cunningham | *CHECKS: A Pattern Language of Information Integrity* http://c2.com/ppr/checks.html |
| Dyson 2004 | *Architecting Enterprise Solutions: Patterns for High-Capability Internet-based Systems*, Paul Dyson and Andy Longshaw, John Wiley and Sons, 2004 |
| Gamma 1995 | *Design Patterns*, Addison Wesley, 1995. |
| Haase | *Java Idioms – Exception Handling*, linked from http://hillside.net/patterns/EuroPLoP2002/papers.html |
| Harrison | Patterns for Logging Diagnostic Messages, Neil B. Harrison |
| Longshaw 2004 | *Patterns for the Generation, Handling and Management of Errors,* Andy Longshaw and Eoin Woods, EuroPLOP 2004. |
| Renzel 97 | *Error Handling for Business Information Systems*, Klaus Renzel, linked from http://hillside.net/patterns/onlinepatterncatalog.htm |


## Acknowledgements

We'd like to thank our EuroPLoP 2005 shepherd, Ofra Homsky for her thorough and valuable feedback during this paper's review process and our original EuroPLOP 2004 shepherd Bob Hanmer for providing very valuable advice on the original paper.

# Appendix: Expected vs. Unexpected and Domain vs. Technical Errors

This pattern language classifies errors as “domain” or “technical” and also as “expected” and “unexpected”. To a large degree the relationship between these classifications is orthogonal. You can have an expected domain error (no funds in the account), an unexpected domain error (account not in database), an expected technical error (WAN link down – retry), and an unexpected technical error (missing link library). Having said this, the most common combinations are expected domain errors and unexpected technical errors.

A set of domain error conditions should be defined as part of the logical application model. These form your expected domain errors. Unexpected domain errors should generally only occur due to incorrect processing or mis-configuration of the application.

The sheer number of potential technical errors means that there will be a sizeable number that are unexpected. However, some technical errors will be identified as potentially recoverable as the system is developed and so specific error handling code may be introduced for them. If there is no recovery strategy for a particular error it may as well join the ranks of unexpected errors to avoid confusion in the support department (“why do they catch this and then re-throw it…”).

Table 1 illustrates the relationship between these two dimensions of error classification and the recommended strategy for handling each combination of the two dimensions, based on the strategies contained in this collection of patterns.

| | ***Expected*** | ***Unexpected*** |
|---|---|---|
| ***Domain*** | • Handle in the application code<br>• Display details to the user<br>• Don’t log the error | • Throw an exception<br>• Display details to the user<br>• Log the error |
| ***Technical*** | • Handle in the application code<br>• Don’t display details to the user<br>• Don’t log the error | • Throw an exception<br>• Don’t display details to the user<br>• Log the error |

**Table 1- Error Handling Strategies**