Patterns for 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. To do so, they should **Keep Exceptions Exceptional**, **Hide Technical Details from Users** and encapsulate the system in a **Big Outer Try Block**. The administrators must be informed when an error occurs and must be given sufficient information about what happened, where it happened and why it happened. If a system has **Split Domain and Technical Errors** and its components only **Log Unexpected Errors** then the level of error information can stay manageable. If distributed systems **Log at Distribution Boundaries** then the overall error information in the system can be reduced and the consequences of individual errors can be tied together using a **Unique Error Identifier**.

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.

This paper sets 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. These patterns 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.

Overview

The patterns presented in this paper form a pattern collection to guide 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 paper are intended to help with these system-wide decisions such as whether to handle domain errors (errors in business logic) and technical errors (platform or programming errors) in different ways. 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 in the collection are illustrated in Figure 1.

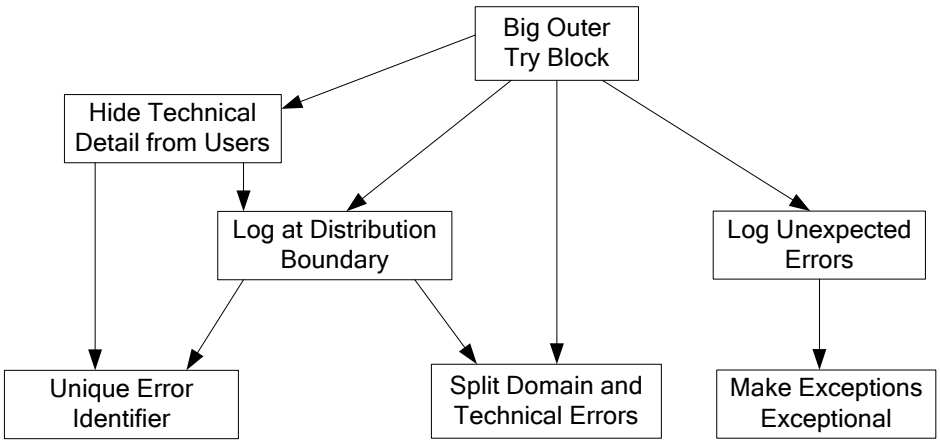

Figure 1 - Error Handling Patterns

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 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 for 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.

At the end of the paper, a set of proto-patterns is briefly described. These are considered to be important concepts that may or may not become fully fledged patterns as the paper evolves.

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.

Context

Domain and technical errors form different "areas of concern". Technical errors "rise up" from the infrastructure - either the (virtual) platform, e.g. database connection failed, or your own artifacts, e.g. distribution facades/proxies. Business errors arise when an attempt is made to perform an incorrect business action. This pattern could apply to any form of application but is particularly relevant for complex distributed applications as there is much more infrastructure to go wrong!

Forces

- If domain code handles technical errors as well as domain ones, it becomes unnecessarily complex and difficult to maintain.
- A technical error can cause domain processing to fail and the system should handle this scenario. However, it can be difficult (or impossible) to predict what types of technical errors will occur within any one piece of domain code.
- It is common practice to handle technical errors at a technical boundary (such as a remote boundary). However, such a boundary should be transparent to domain errors.
- For some technical errors, it may be worth taking certain actions such as retrying (e.g. retry a database connection). However, such an action may not make sense for a domain error (e.g. no funds) where the inputs remain the same.
- As part of the specification of a system component, all of the potential domain errors originating from a domain action should be predictable and testable. However, changes in implementation may vary the number and type of technical errors that may possibly arise from any particular action.
- Technical and domain errors are of interest to different system stakeholders and will be resolved by members of different stakeholder groups.

Solution

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

Implementation

Errors in the application should be categorized into domain errors (aka. business, application or logical errors) and technical errors. When you create your exception/error hierachy for your application, you should define your domain errors and a single error type to indicate a technical error, e.g. SystemException (see Figure 2). 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).

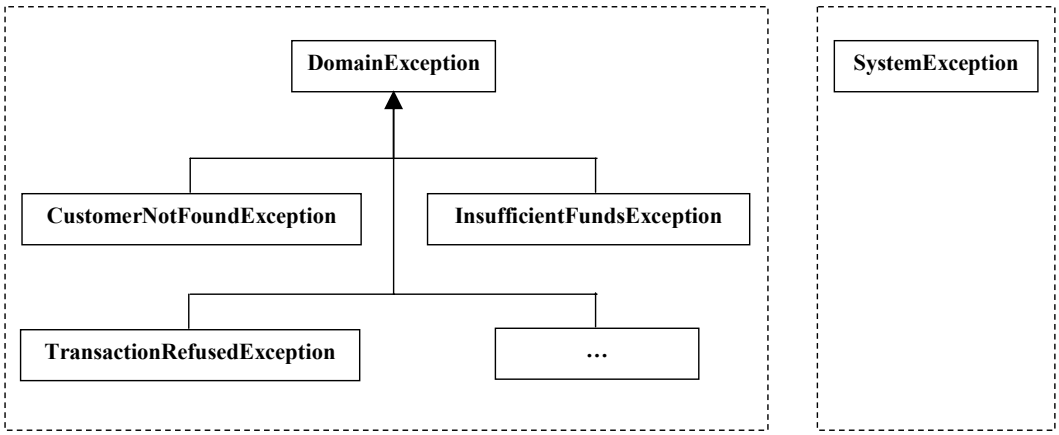

Figure 2 – Split Domain and Technical Exception Hierarchy

Design and development policies should be defined for domain and technical error handling. These policies should include:

- A technical error should never cause a domain error to be generated (never the twain should meet). When a technical error must cause business processing to fail, it should be wrapped as a SystemError.
- Domain errors should always start from a domain problem and be handled by domain code.
- Domain errors should pass "seamlessly" through technical boundaries. It may be that such errors must be serialized and re-constituted for this to happen. Proxies and facades should take responsibility for doing this.
- Technical errors should be handled in particular points in the application, such as boundaries (see *Log at Distribution Boundary*).
- The amount of context information passed back with the error will depend on how useful this will be for subsequent diagnosis and handling (figuring out an alternative strategy). You need to question whether the stack trace from a remote machine is wholly useful to the processing of a domain error (although the code location of the error and variable values at that time may be useful).

As an example, consider the following exception definitions:

```
public class DomainException extends Exception
{
  ...
}
Public class InsufficientFundsException extends Exception
{
  ...
}
public class SystemException extends Exception
{
  ...
}
```

A domain method skeleton could then look like this:

```
public float withdrawFunds(float amount)
             throws InsufficientFundsException, SystemException
{
  try
  {
    // Domain code that could generate various errors
    // both technical and domain
  }
  catch (DomainException ex)
  {
    throw ex;
  }
  catch (Exception ex)
  {
    throw new SystemException(ex);
  }
}
```

This method declares two exceptions: a domain error – lack of funds to withdraw – and a generic system error. However, there are many technical exceptions that could occur (connectivity, database, etc.). The implementation of this method passes domain exceptions straight through to the caller. However, any other error is converted to a generic SystemException that is used to wrap any other (non-domain) errors that occur. This means that the caller simply has to deal with the two checked exceptions rather than many possible technical errors.

Consequences

- The business error handling code will be in a completely different part of the code to the technical error handling and will employ different strategies for coping with errors.
- Business code needs only to handle any business errors that occur during its execution and can ignore technical errors making it easier to understand and more maintainable.
- Business error handling code can be clearer and more deterministic as it only needs to handle the subset of business errors defined in the contract of the business methods it calls.

- All potential technical errors can be handled in surrounding infrastructure (server-side skeleton, remote façade or main application) which can then decide if further business actions are possible.
- Domain errors are passed through infrastructure code – possibly by marshaling and unmarshaling them across the infrastructure boundary (typically a distribution boundary).
- Different logging and auditing policies are easily applied due to the clear distinction of error types.

Related Patterns

- Technical and domain errors should be treated differently at distribution boundaries as defined in *Log at Distribution Boundary*
- Unless they are handled elsewhere in the system, both technical and domain errors should be handled by a *Big Outer Try Block*.
- It is common to apply the proto pattern *Single Type for Technical Errors*
- A more general form of this pattern is described in *Exception Hierarchy* [Renzel 97]
- The use of a domain hierarchy in Java is also discussed in the *Exception Hierarchy* idiom in [Haase]
- The *Homogenous Exception* and *Exception Wrapping* Java idioms in [Haase] show how you might implement the a single SystemException in Java.

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.

Context

Multi-tier systems, particularly those that use a number of distinct technologies in different tiers.

Forces

- You could propagate all the error information back to original calling application where it could be logged by a Big Outer Try Block but the complete set of error information is bulky and may include platform-specific information.
- Technical error information needs to be made easily accessible to the technology specialists (such as operating system administrators and DBAs) who should be able to resolve the underlying problems.
- When administrators come to resolve problems that technical errors reveal, they will need to access the error logs used by other parts of the system infrastructure as well as using the information in the error logged by the application. In order to facilitate this, technical errors need to be recorded in a log that is easily accessible from the same place as the infrastructure's error logs.
- Each technology platform has its own formats and norms for error logging. In order to fit neatly into the technology environment, it is desirable that the new system uses an appropriate logging approach in each environment.
- To correctly diagnose technical errors that occur on a particular system, extra technical information is often required about the current runtime environment (such as number of database connections open) but adding the additional code needed to recover and record this information to the various layers of application code would make such code significantly more complex.
- The handling of errors should not impact the normal behaviour of the system unnecessarily. To reduce any impact it is desirable to avoid passing large quantities of error information around the system.

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.

Implementation

Implement a common error-handling library that enforces the system error handling policy in each tier of the application. The implementation in each tier should log errors in a form that technology administrators are used to in that environment (e.g. the OS log versus a text file).
The implementation of the library should include:

- Interfaces to log technical and domain errors separately; and
- A generic SystemError class (or data structure) that can be used to pass summary information back to the caller.

The library routine that logs technical errors (e.g. "*technicalError()*") should:

- Log the error with all of its associated detail at the point where it is encountered;
- Return a unique but human readable error instance ID (for example, based on the date such as "20040302.12" for the 12th error on 2nd March 2004); and
- Capture runtime environment information in the routine that logs a technical error and add this to the error log (if appropriate).

Whenever a technical error occurs, the application infrastructure code that catches the error should call the *technicalError* routine to log the error on its behalf and then create a SystemError object containing a simple explanation of the failure and the unique error instance ID returned from *technicalError*. This error object should be then returned to the caller as shown below:

```
...

public class AccountRemoteFacade implements AccountRemote
{
  SystemError error = null;

  public SystemError withdrawFunds(float amount)
            throws InsufficientFundsException, RemoteException
  {
    try
    {
      // Domain code that could generate various errors
      // both technical and domain
    }
    catch (DomainException ex)
    {
      throw ex;
    }
    catch (Exception ex)
    {
      String errorId = technicalError(ex);
      error = new SystemError(ex.getMessage(), errorId);
```

```
    }
  }
  return error;
}
```

If a technical error can be handled within a tier (including it being "silently" ignored [See proto-pattern Ignore Irrelevant Errors] - except that it is *always* logged) then the SystemError need not be propagated back to the caller and execution can continue.

Consequences

- Only a required subset of the technical error information is propagated back to the remote caller – just enough for them to work out what to do next (e.g. whether to retry).
- Technical error information is logged in the environment to which it pertains (e.g. a Windows 2000 server) and in which it can be understood and resolved.
- The technical error information is logged in a similar way to (and potentially in the same place as) other system and infrastructure error information. This may make it easier to identify the underlying cause (e.g. if there are lots of related security errors alongside the database access error).
- Using local error logging mechanisms makes the logs much easier for technology administrators to access using their normal tools, but means that the approach used in each tier of the system may be different.
- The logging mechanism for technical errors can decorate the error information with platform-specific information that may assist in the diagnosis of the error.
- One error can cause multiple log entries on different machines in a distributed environment (see *Unique Error Identifiers* pattern).

Related Patterns

- Implementing *Split Domain and Technical Errors* before *Log at Distribution Boundary* makes implementation simpler, as it allows the two types of error to be clearly differentiated.
- *Unique Error Identifiers* are needed if you want to tie distributed errors into a *System Overview* [Dyson 2004].

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 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 when the original error 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.

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 he hosts in the system. 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 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.

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.

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.

Context

A distributed system with a largely "lay" user community, probably using graphical user interfaces. The interface is likely to be very simple: possibly even a "kiosk style" interface. Users are mostly on remote sites and will not do much to report errors if they can work around them.

Forces

- If an in-depth error report, particularly for a technical error, is presented to an end user, they are unlikely to be able report its content in enough detail for the underlying problem to be unambiguously identified and so the details of the error are likely to be lost.
- If technical errors are presented to users on a regular basis, they will start to ignore them rather than to go through the process of trying to report them and knowledge of the existence, as well as the details, of these errors will be lost entirely.
- Members of the Support Staff need to be able to associate user problem reports with logged error information but detailed error information such as unique error numbers can be very big and it all looks the same to an end user, making it difficult to report.
- We want to avoid having to write code to handle technical errors at multiple layers of the application but this opens up the risk that such errors will "leak" through to the user.

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.

Implementation

In the system's ultimate client, wrap the top-level invocation of the system in a *Big Outer Try Block* that will catch any error – domain or technical – propagating up from the rest of the system. The *Big Outer Try Block* should differentiate between technical errors (such as databases not being available) and domain errors (such as performing business process steps in the wrong order) as suggested in *Split Domain and Technical Errors*.

Technical errors should be logged for possible use by technical support staff and the user should then be informed that something terrible has happened in general terms, making it clear that what has happened is not related to their use of the system.
A domain error that reaches the *Big Outer Try Block* is probably a failure in the design of the user interface that resulted in an unanticipated business process state being reached and as such should be treated as a system fault. In such cases, again the error should be logged and a user-friendly message displayed, but in this case the message can include details of the problem encountered, as these details are likely to be meaningful to the user since they relate to the business process that they were performing.
Finally, a totally unpredictable error (such as an exception indicating a resource shortage such as having run out of memory) that reaches the *Big Outer Try Block* is some form of internal or environmental error that could not be handled at a lower level. As with a technical error, a generic error should be displayed to the user and the details of the error logged locally.
An example of the structure of a *Big Outer Try Block*'s implementation is shown in the following code fragment:

```
public class ApplicationMain
{
  ...
  public static void main(String[] args)
  {
    try
    {
        ApplicationMain m = new ApplicationMain() ;
        m.initialize() ;
        m.execute() ;
        m.terminate() ;
    }
    catch(AppDomainException de)
    {
      // Domain exceptions shouldn't get to this level as
      // they should be handled in the user interface.  If
      // they get here, report the text to the user and
      // log them in a local log file
    }
    catch(AppTechnicalException te)
    {
      // Technical exceptions here are probably user interface
      // problems.  Display a generic apology and log to a
      // local log file
    }
    catch(Throwable t)
    {
      // Other exception objects must be internal errors
      // that could not be caught and handled elsewhere.
      // Display a generic apology and log to a local log file
    }
  }
}
```

Consequences

- Error information is never lost because of unexpected errors propagating to the edge of the system and "leaking out". The error information is always captured in its entirety to allow it to be retrieved for support and diagnostic purposes.
- Users are never surprised by an application simply stopping or crashing, but are always informed that something has gone wrong in a user comprehensible form.
- If the application does fail in an unexpected way, it always handles this condition in a consistent manner.
- Other parts of the system may have simpler error handling as they do not need to include handling for totally unpredictable errors.

Related Patterns

- Implementing *Split Domain and Technical Errors* makes the implementation of *Big Outer Try Block* simpler because the different types of error can be easily differentiated.
- This pattern can be combined with the *Hide Technical Details from Users* pattern in order to ensure that suitable messages are reported to the user when the *Big Outer Try Block* is triggered.
- This pattern can be combined with *Unique Error Identifier* in order to ensure that errors logged by the *Big Outer Try Block* can be clearly identified.
- A Big *Outer Try Block* is a form of *Default Error Handling* [Renzel 97]
- This concept is also mirrored in the Java idiom *Safety Net* in [Haase]

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 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.

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.

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 to allow the user to easily report what has happened, via some form of helpdesk. Ideally, the user reporting of the error should be automated in some way – perhaps using desktop email automation. 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

- 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.

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.

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.

Context

Where systems are created in organizations with complex domain processing or processes with a large number of routinely expected error conditions that the processes specify the response to.

Forces

- The system should report errors when they occur so that they can be investigated and fixed but it is easy for serious errors to be hidden under large numbers of spurious or trivial problems. It should be obvious when operator intervention is required.
- If all possible error conditions, including those routinely encountered during normal operation, are reported then log management becomes much more difficult due to the speed at which the error logs fill up.
- Recording lots of errors increases the amount of logging code and the number of error messages that need to be managed, which reduces the maintainability of the system.

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.

Implementation

Throughout the system's implementation, use two distinct error handling approaches for expected and unexpected errors:

- Log unexpected errors according to the other patterns, such as *Log at Distribution Boundary*, and put in place a process that ensures the error triggers operator intervention to resolve the situation.
- Do not log expected errors, but handle them as part of the system's normal operation. This may be done in the code itself, maybe by trying different domain logic that may be able to handle the given inputs or scenario or creating alternative domain entities.

  Alternatively, the application may interact with the user, inform them of the problem (in appropriate terms – see *Hide Technical Detail from Users*) and prompt them to re-start part or all of the current operation.

By following these principles, errors such as "could not connect to database" are not hidden by hundreds of routine error conditions such as "no such product code" (perhaps caused by a user misreading a code from a piece of product packaging). As the former error is a significant error requiring investigation, while the second is an expected error condition, the former would be logged and the latter handled algorithmically by the business logic, without logging the condition.

One variation on this approach is to log different types of error message to different places. For example, in terms of the application itself a user failing to authenticate may not be worth recording. However, from the system's point of view (i.e. the operating system) the security policy may require all failed authentications to be logged. This is usually resolved by logging different types of errors to different logs, such as the application event log and security event log provided under Windows. Such partitioning allows different logs to be created to serve the needs of different areas of concern.
Another example of this is where knowledge of the patterns in which errors occur would be of interest to developers – large numbers of failed searches at a search engine site may indicate a usability problem. However, such errors are not of interest to the operations team who are responsible for keeping the system running. In this case, the expected errors could be logged to a different location where they will not interfere with the operational errors but can be retrieved later by the development team for further analysis.
A second variation is to log different types of error message in one location but to mark each log message with one or more attributes that allow a set of filters to be created to provide the ability to extract various subsets of the log content on demand to support different uses (such as error monitoring versus usability analysis).

Consequences

- Errors that appear in logs always indicate exception conditions and so can be used to initiate support and diagnostic activities.
- Spurious messages indicating that expected conditions have occurred do not prevent easy recognition of the occurrence of exceptional conditions.
- Logs do not quickly fill up with spurious messages created as a result of normal operation.
- Application code is simplified as a result of the reduction in the number of log messages that need to be produced.

Related Patterns

- You need to ensure that the correct distinction is made between expected errors and exceptional occurrences as described in *Make Exceptions Exceptional*.
- It may be helpful to classify errors as "domain" or "technical" errors, as described in *Split Domain and Technical Errors*.
- An approach such as *3 Category Logging* [Dyson 2004] can help to make a log filterable.

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.

Context

Any situation where a language with exception handling built into it is in use.

Forces

- An application should be designed to handle and recover from most domain errors but some unexpected errors will always occur. Examples of the latter include incorrect or missing application data in the database and incorrect or missing values in configuration files.
- The code paths for handling "recoverable" errors and "unrecoverable" errors are usually quite distinct so they should be easily differentiated.
- Large numbers of exceptions generated cause problems for the consumer of a class/method – especially in a language that uses checked exceptions.
- We want to avoid convoluted code and algorithm distortion when routine error conditions (such as "end of list") are encountered.

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.

Implementation

When designing the interfaces in your system you should classify errors into two types:

- Conditions that will occur routinely in standard algorithms, which should be indicated by returning a reserved return value. This could be a null pointer, an empty list or a specific return code (E_INVALID). An example of this would be returning an empty list from a search operation that did not match any items.

- Conditions that will only occur due to unexpected errors, which should be indicated by raising language exceptions. Examples of such conditions include those caused by underlying platform or network failure (cannot connect to database), incorrect configuration (bad database connection string) or bad application data (customer id – not name – could not be found).

Errors of the first type will be handled as part of the standard business logic in the system. On the other hand, errors of the second type will normally be handled by a combination of logging and exiting the current code block via an exception path.

It is worth briefly exploring the differences and the blurring of the boundaries here through an example. Consider a component in a retail system that offers out two methods to look up product information. One of these methods allows you to look up products either by keyword or wildcard text string and returns a list of matching products. The other method requires a numeric product code such as a barcode and returns the single product matching that code. The component is backed by a database containing all the products stocked by the retailer.

The search by keyword/wildcard has no guarantee of finding a matching product. Typically, the keyword/wildcard will be entered by a user and so could be subject to all forms of data problems such as mis-spelling or unrealistic expectations (e.g. entering "Elton John" when the retailer just sells food – not CDs). Hence, semantically you could expect no products to be returned – this is an expected business condition, however unhelpful it is to the user of the calling application. Having said that, the user can always get the answer they want by trying again – providing sensible input to the search.

On the other hand, there is more of a semantic implication that the method that requires a product code should find something. Unless users of the system are prone to scanning in barcodes from random products they bring into work, any product scanned in store should be in the database: you should not be able to provide a code that cannot be found. In this case, you could justifiably throw an exception as the only way this condition can occur is if there is a problem with the data in your database. Not only can the user not get the right answer by re-scanning the product (same answer each time…), but in terms of the system this situation needs resolving (i.e. the data in the database needs correcting).

Finally, in either case if the component cannot connect to the database for whatever reason a technical exception should be raised (indeed, the underlying platform will probably raise one for you).

Consequences

- Application code is simplified as it does not need to include exception handling constructs within normal algorithms.
- Exceptional conditions in code can all be treated as abnormal situations requiring error handling and as such can be handled via a uniform strategy.

Related Patterns

- Expected errors should not be logged, as described in *Don't Log Business Process Errors*, but unexpected errors – whether technical or domain exceptions – should be logged.

- Ward Cunningham's CHECKS pattern language for information integrity provides a great deal of guidance relating to the design of data validation in user interfaces (i.e. "Type 1" errors in this pattern's Implementation section).

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 96Pattern-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

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 shepherd, Bob Hanmer, for his thorough and valuable feedback during this paper's review process and the members of the OT2004 workshop at which this paper was first presented.

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 • Display details to the user • Don't log the error	• Throw an exception • Display details to the user • Log the error
Technical	• Handle in the application code • Don't display details to the user • Don't log the error	• Throw an exception • Don't display details to the user • Log the error

Table 1- Error Handling Strategies