Error Handling and Diagnosability

ACCU 2008

Tony Barrett-Powell
tony.barrett-powell@oracle.com

Introduction

- A look at error handling and diagnosability
- Some thoughts about costs
- Some ideas about problems and approaches
- Some useful design patterns for error handling

Who am I?

- A developer at Oracle working on Business Intelligence tools
- ACCU committee member and web site editor
- I've seen too much poor error handling
  - It's makes fault finding difficult
  - It's reduces the value of software to the customer and to the vendor

A cynical view

- Much software has “primitive” error handling
  - Throwing exceptions “into the surprised face of the user” [stob]
  - Yet it is commercially viable
  - It must be low on the list of customer needs
- Errors don't happen (or all that much)
  - Re-enforced optimism?
  - When errors occur it is too difficult for a developer to understand how to handle them?

A cynical view (cont.)

- Darwinism at work?
  - Software with “advanced” error handling to expensive to write?
  - Or too late to market?
- Maybe it is no more than an aspect of the “big ball of mud” pattern [foote]

Cost of development and support

- Software with poor error handling will cost more to support and may be harder to change than with good error handling
- Adding error handling too late is likely to increase entropy and may increase costs
- Adding error handling too early (in the prototype or expansion phase) may increase costs [foote]

The need for error handling

- Criticality of error handling in software?
  - Software for a single user with short life span is unlikely to need much in the way of error handling
  - Complex software used by many 1000s of users is likely to need good error handling

Dimensions of criticality

- Size of user base
  - A small user base can be supported individually by developers, but this is impossible with many 1000s of customers
- Capability of the user base
  - A technically aware user base is likely to be able to resolve faults without help, a non-technical user base will require more from the software

Dimensions of criticality (cont.)

- Complexity
  - For any measure of complexity (code size, ability to debug, understandability, team size) low complexity allows simple diagnosis of faults, high complexity in any measure makes this much more difficult
- Urgency
  - If the software is critical to a customer or user base then timely and accurate diagnoses of faults is important as this maximizes the availability of the software

Dimensions of criticality (cont.)

- Scale
  - The amount of data consumed or created, or the number of components in a deployment when these are large can make diagnoses of faults difficult
- Competition
  - a software product should be no worse than the competition to be competitive, better could be an advantage

Dimensions of criticality (cont.)

- Lifetime
  - If the software will be supported for many years the cost of this will far outweigh the cost of initial development

The need for error handling

- Problematic values for the above dimensions indicate the need for good error handling
  - To reduce overall cost of development and support
  - To reduce risk of failure
  - To allow continued development
- The values of these dimensions may change during the lifespan of software
  - Generally increasing
  - When crossing the chasm [moore]

Errors and Faults

- I've been using the terms *errors* and *faults*
- Errors to mean the consequence of faults
- Faults to mean the source of errors
  - Though cause may be a better term as it carries less moral baggage
- We see and handle errors in our code
- We determine and diagnose faults from the errors we see

Some Definitions

- What is an error?
  - A user error
    - The user tried to book a holiday starting last week, entered their account number incorrectly, these are domain errors and should be handled gracefully by the software
  - A system failure
    - The user attempted a completely reasonable request but the system could not service the request, for example the database refused the creation of a connection or the network was down. The user could try again and it might work, indeed we might want to do this automatically in the software

Definitions (cont.)

- What is an error?
  - A software bug
    - The user attempted a completely reasonable request but the system could not service the request, there was a programming fault in the software. There is nothing the user can do, a repeated request will fail in the same way.

Definitions (cont.)

- What is diagnosability?
- The attribute of software which allows diagnoses of faults in a software system
- Practically, it is the information about an error
  - What, why and when the error happened
  - What was being done
  - How to fix it

Reporting Faults

- Do nothing
  - Or even worse actively hide it
  - At best it'll mislead diagnosis
  - At worst it'll cause a crash
- Throw an exception
  - If your language supports it
  - Great for separating error handling and normal flow of execution
  - Care needs to be taken with resources and exception safety

Exception guarantees

- Anyone not aware of these?
- Basic guarantee
  - Resources not leaked
  - State will be consistent but may be changed
- Strong guarantee
  - State is unchanged
- Nothrow guarantee
  - Never throws an exception

Reporting faults (cont.)

- Return an error value
  - Messy if a value is already returned
  - Can be ignored
    - Though this can be handled in C++ using [jagger]
- Set an error status
  - Shared global status variable
  - Used by the C standard library: errno
- Raise a signal
  - An operating system facility, basically an interrupt event, requiring a registered handler

Reporting faults (cont.)

- Crash
  - Sometimes it's so bad its best to fail fast and get the core file
- Consistency is important
  - A mix of detail level
    - for error reported to the end user will be confusing
    - in logging will make diagnosis difficult
  - Providing the right information about faults to the right audience is important

Reporting faults (cont.)

- Significant roles to consider when reporting
  - End user
  - Support
    - Vendor's support services
    - Administrator of the system
  - Development
    - Testers
    - Developers

What should we include?

- A descriptive string
- Where the error occurred, class, file, line
- Stack trace, if available
- Possibly some error code
- A name for the error
- Information about what was being done at each level of abstraction
- The configuration of the system and the environment

Levels of abstraction

- Reporting the lowest level error
  - Breaks encapsulation – it exposes the implementation
  - Does not help diagnose the problem
- Each level of abstraction must provide its understanding
  - Socket error -> Database communications error -> Withdraw funds error -> ... -> Pay bill error
- We should not discard the lower level information

Detection and Cause

- When an error is detected
  - This may be at the location of the fault
  - Though equally, it may be separate from the fault by time, space or both
- We should aim to reduce the space and time from fault to detection
  - Push business logic close to the significant borders of the system
  - Detect system faults near to use
  - But is not always possible

Diagnosability and logging

- When diagnosability is discussed the conversation always ends with logging
- It seems to be the best way to provide the information needed to diagnose errors, especially those that are hard to reproduce
- We should use our experience in adding diagnostic logging during development to guide us

Logging

- What if logging had no performance or storage penalties? We could
  - log every intermediate state in the system
  - reconstruct the context for every error whenever or wherever it was detected
  - build tools to explore back through the log and diagnose faults – a diagnosis debugger?
- Of course this isn't the case, but maybe we can implement some aspects

Infinite memory/Infinite log

- A garbage collected language is based on the concept of infinite object lifetime
  - Objects notional exist forever
  - Those beyond reach can be quietly deleted
- Could data carry its own context?
  - Where it can from, how it was processed by the system - its lineage
  - If the data is found to be at fault we have the information to diagnose the fault
  - lineage notionally exists forever, but is discarded when it is no longer needed

Play State

- We could build context information as a request passes through the system
  - Ready to be added to an exception as it passes through the code
  - Disposed of at the completion of the request
  - Much like the diagnostic logging we all add for repeatable errors, but ready for any errors, even the most infrequent
  - This context can be logged for attention of support or developers

Pushing the Play State

- Play State may mean we add more error handling code than is otherwise necessary
- We could push the context down the stack instead
  - Using a Encapsulated Context Object [henney]
- When an exception is thrown the semantic context is available
  - This is more useful than a stack trace

Dynamic logging level

- We can allow the software to tune the amount and level of logging or this can be controlled manually
- Could make use of the Encapsulated Context Object [henney]
- The software reduces logging in flows where no errors have occurred, increases when an error is detected
- A non-deterministic impact on performance
- Not so good for infrequent or one-off errors

Component Failure

- A component may crash or the hardware fail leaving no record of the initial error
  - We have a gap in our history making diagnosis more difficult
- We can add remote logging
  - But this adds new failure modes to the system if it is transactional
  - We can push logging in the background without caring if it arrives
  - Consider Offline Reporting [dyson+]

Category Logging

- We can use the 3-Category Logging from Architecting Enterprise Solutions [dyson+]
  - Debug data
    - Execution trace information, methods and parameters
  - Information
    - Timeouts and missing data
  - Error data
    - Database connection failure

Distributed components and Logs

- If we have many distributed components especially those that are load balanced it will be difficult to gather the information about an error
- We could introduce a System Overview [dyson +] to transparently gather the logs into a aggregated view which can be analyzed easily
- We could use the System Overview to package individual errors as part of a incident report

Logging and Security

- Sensitive information may be logged
- Use Information Obscurity [dyson+] to grade the sensitivity of data
- Logging mechanisms should use a suitable encryption or obfuscation technique to protect this information in the logs
- It could be filtered out, but might be important in the diagnosis of faults

Error Handling Policy

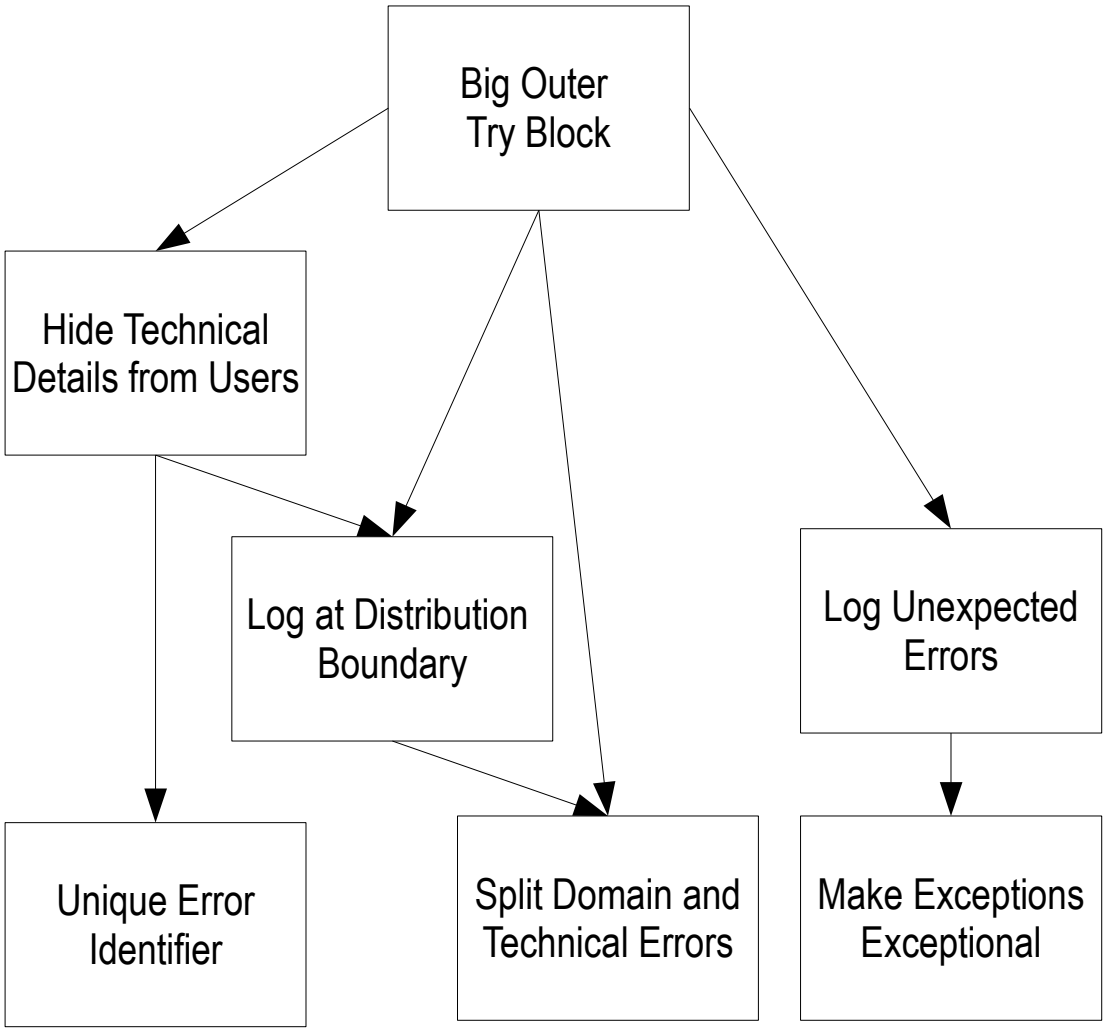
- Each project should have one
- Guidance on the approaches to use for error handling
  - Informs best practice
  - Improves consistency
  - Provides information on available frameworks
- Should be enforced by review, much like a house coding style

Error handling patterns

- Patterns for Generation, Handling and Management of Errors
- A pattern language collected by Eoin Woods and Andy Longshaw
- Workshopped at SPA2004 and EuroPLoP 2005
- The following is a abridged version of these patterns

Error Handling Patterns Language

- Make Exceptions Exceptional
- Split Domain and Technical Errors
- Unique Error Identifier
- Log at Distribution Boundary
- Log Unexpected Errors
- Hide Technical Details from Users
- Big Outer Try Block



Make Exceptions Exceptional

- Problem
  - Exceptions are a good thing, but use for expected error conditions reduces the ability to understand calling code
  - “recoverable” and “non-recoverable” errors are handled in different ways

Make Exceptions Exceptional

- Solution
  - Only use exceptions to indicate runtime problems
  - Conditions that occur routinely should be handled by a suitable return value
  - Error conditions that only occur due to unexpected errors should be indicated by raising an exception
  - Consider database lookup by wildcard or lookup by key as examples of the above cases
  - Exceptional conditions should be treated as abnormal situations and handled in a uniform manner

Split Domain and Technical Errors

- Problem
  - Error conditions related to the domain and those related to the technical implementation are different concerns. Handling these in the same code makes it harder to understand and maintain
  - “recoverable” and “non-recoverable” errors are handled in different ways

Split Domain and Technical Errors

- Solution
  - Errors should be categorized into domain and technical errors, this should be reflected in the exception hierarchy
  - Technical errors should not cause a domain error

Unique Error Identifier

- Problem
  - If an error in a distributed system causes knock-on errors understanding the cause of errors is difficult
  - Knock-on errors can be correlated but this takes both skill and effort
  - Load balanced systems results in non-deterministic paths between components making it difficult to determine the location of error logs for a particular fault

Unique Error Identifier

- Solution
  - Generate a unique error identifier when the original error occurs which is used consistently for all knock-on errors
  - Unique identifiers should be unique across hosts, so a GUID or UUID is an obvious solution

Log at Distribution Boundary

- Problem
  - Details of technical errors make little sense outside the boundary of the component in which they occur
  - Propagation of technical error details results in these details appearing in a context far removed from the context of the original error, such as an end-user browser
  - Technical error details should be available to those best able to solve the fault
  - Each platform has specific formats of standards for error logging

Log at Distribution Boundary

- Solution
  - When technical errors occur log them on the system where they occur, and return a generic system error
  - This allows calling code to handle the error appropriately, but does not require the system-specific information to passed back through the system

Log Unexpected Errors

- Problem
  - Much domain code includes handling of exceptional conditions and handles these according to understanding in the domain (for example an faulty transaction being rejected)
  - If these domain errors are logged this pollutes the contents of the logs with content which is not relevant to identifying and resolving system problems

Log Unexpected Errors

- Solution
  - Mechanisms for expected and unexpected errors should be separate
  - Domain error conditions should be handled in the code or by the user
  - Unexpected error conditions should be logged and therefore viewed as requiring investigation
  - Alternatively, Category Logging [dyson+] should be considered

Hide Technical Details from Users

- Problem
  - The technical details of errors are typically of no interest to end users and incomprehensible
  - Exposed system details may overly concern end-users, decrease confidence in the system and increase support overhead
  - Technical errors have information useful to support staff but is little value to end users

Hide Technical Details from Users

- Solution
  - Implement a standard mechanism for reporting technical errors to end users
  - Display a user friendly message to inform the user that something bad has happened in general terms, which is nothing to do with their use of the system, possibly providing some automated reporting mechanism
  - Full details of the error should be logged for support staff

Big Outer Try Block

- Problem
  - Unexpected errors can occur in any system
  - Truly exceptional conditions are rarely anticipated in the design of a system and will propagate to the edge of the system
  - If not handled valuable information about the error may be lost, leading to problems in diagnosis of the underlying problem

Big Outer Try Block

- Solution
  - Implement a Big Outer Try Block at the edge of the system to catch and handle errors that cannot be handled by other tier or components of the system
  - The error handling block can report errors in a consistent way at a level of detail appropriate to the user
  - Full information about the error can be logged for the attention of support staff for diagnostic pusposes

Summary

- I want to see mature error handling in software
- Understanding the error handling techniques of a language is not enough
- We should be able to create a suitable and consistent error handling style for each software project we work on
- The patterns I've mentioned provide some useful solutions to be considered when embarking on a project

References

[stob] http://www.regdeveloper.co.uk/2006/01/11/exception_handling

[foote] http://www.joeyoder.com/papers/patterns/BBOM/mud.html#BigBallOfMud
and http://en.wikipedia.org/wiki/Big_ball_of_mud

[longshaw+] http://www.blueskyline.com/Patterns_files/ErrorPatternsPaper.pdf

[henney]
http://www.two-sdg.demon.co.uk/curbralan/papers/europlop/ContextEncapsulation.pd

[moore] http://en.wikipedia.org/wiki/Crossing_the_Chasm

[goodliffe] Code Craft: The Practice of Writing Excellent Code, Pete Goodliffe

[griffiths] If problems arise, C Vu 14.2 http://accu.org/index.php/journals/1160

[dyson+] Architecting Enterprise Solutions: Patterns for High-Capability Internet Systems, Paul Dyson and Andy Longshaw

[weinberg] Quality Software Management, Volume 1 Systems Thinking, Gerald M. Weinberg