Exceptional design

Hubert Matthews, January 2007

This article describes the approach I have taken to designing applications that I have written over the last few years. It is also a distillation of the techniques I have been teaching on C++ courses for some years. It is not meant to be the one and only true way of designing using exceptions, but rather a report on techniques I have found useful. The second part of the article explores why this approach isn't universal and some of the barriers – psychological and technical – to its adoption.

In a nutshell, the core design concepts I use are:

- Many throws, few catches
- Placement of catches is determined by recovery points, which are based around business requirements not technical considerations
- Catches should also be placed at module boundaries
- Implement the strong exception safety guarantee whenever reasonable
- Make intermediate code exception-neutral
- Log at the point of detection, recovery (if any) at the point of handling
- Use local control structures in preference to throwing an exception
- Use standard exceptions or subclasses thereof
- Use as few subclasses as possible, ideally zero or one
- Add additional information to the exception class for multi-level recovery and for business and technical context
- Don't nest exceptions

These techniques are primarily focused on C++, the language I code in the most. C++ has a cleaner exception model than Java and C# as well as deterministic destruction (i.e. destructors instead of try/finally blocks), both of which make the use of exceptions easier and more elegant.

First step

The first thing I do when writing a C++ program is to place a double try/catch block in main():

```
int main()
{
      try {
            // main body of code
      } catch (const std::exception & e) {
            cerr << "Caught exception: " << e.what() << endl;
            return 1;
      } catch (…) {
            cerr << "Caught unknown exception" << endl;
            return 1;
      }
}
```

This ensures that no exception can propagate out of main() and thereby cause the program to terminate unceremoniously (fragile programs that disappear without a trace are not popular!). It is an example of the maxim "catches at module

boundaries". In this case the boundary is between my program and the host operating system (indirectly via the run-time library), which expects a return code. The catch(…) acts as a catch-all (literally) for other exceptions. These may be caused by an errant throw (for example someone throwing a plain integer or a string) or on earlier versions of Visual C++ errors such as access violations and divide-by-zero errors were translated into anonymous C++ exceptions using Windows' structured exception handling (SEH). On Unix, the g++ compiler can be persuaded into turning signals into exceptions, but again this is architecture and processor specific.

This universal "last gasp" handler therefore protects against these anonymous exceptions and adds perceived robustness to the program. One of the programs I wrote and maintained for 8 years on Windows used a third-party library that would occasionally cause access violations, so this catch(…) technique was very useful. Try/catch blocks also need to be placed at the top of every thread function and probably around any code that executes in button handlers in GUI applications (as exceptions cannot propagate across GUI message pumps). In production code more error logging than this would be appropriate (and perhaps even a core dump on Unix).

Recovery strategy

The next step is to determine what the recovery strategy for the program should be. This can range from a graceful shutdown for simple end-user programs through partial completion all the way up to resilient recovery, retry or failover for long-running server programs. Recovery is a business-requirements issue more than it is technical design. The key question to ask is "what should happen now?" rather than "what can we do?". Sometimes recovery is multi-layered with some classes of error being handled at an intermediate level and others being escalated to a higher layer. As an example, one of the applications I wrote had two intermediate layers: the first layer was a list of command scripts to run and the second layer was the scripts themselves. There was a recovery point (i.e. a try/catch pair) around each script line invocation that enabled the script to continue despite an errant line, with a second recovery point around each script file to allow for missing or inaccessible script files. The "last gasp" handler at the outer layer was used to catch errors such as memory allocation errors.

In order to create such a layering, it was necessary to use a custom exception class instead of one of the standard subclasses of std::exception. This custom class inherited from std::exception and added an additional severity flag:

```
class AppException : public std::exception {
public:
      enum Level { warning, severe, fatal };
      AppException(Level level, const std::string & msg)
            : std::exception(msg), level(level) {}
      const Level level;
};
```

This extra level information allows intermediate catch handlers to choose whether to recover or rethrow:

```
try {
      // inner-level code
} catch (const AppException & e) {
```

```
      if (e.level == AppException::warning)
            Log(e);     // "recovery"
      else
            throw;      // appeal to a higher authority
}
```

The severity level for each exception is decided at the call site when the exception is thrown:

```
if (FileIsNotAccessible(filename))
      throw AppException(AppException::severe,
                         "Can't open file " + filename);
```

With such a structure in place handling errors becomes conceptually much easier as if a subroutine detects an error it cannot handle locally then it can just throw. Handling errors locally is of course preferable and normal control structures should be given preference over the use of exceptions.

An alternative approach is to use a very limited exception hierarchy with one subclass per error level. This allows the multi-level catch handler shown above to use the type of the exception to catch only certain error levels:

```
class WarningException : public std::exception {};
class SevereException : public std::exception {};

try {
      // inner-level code
} catch (const WarningException & we) {
      // handle only warnings, let severe exceptions propagate
      // to the next level up
      Log(we);
}
```

The effective difference between these two approaches is minimal; both achieve the same effect. The catch handler is effectively identical and the throw site is also the same in all but syntax.

I use a single exception class rather than an application-specific exception hierarchy as I have found little need to differentiate between different types (rather than different severities) of exception when handling them. Doing so would require a whole chain of catch handlers that rapidly degenerates into the equivalent of an if-else-if chain testing against types:

```
try {
      // code
} catch (const SubException1 & e) {
      // handle exception case 1
} catch (const SubException2 & e) {
      // handle exception case 2
} catch (const SubException3 & e) {
      // handle exception case 3
}
```

There are a number of problems with such code. First, the order of the catches is now important as more derived classes must be caught before their bases. Second, adding

a new exception type requires the catch handlers to be updated, which is a sign of undue coupling. Third, I can think of little significant difference from a recovery perspective that justifies the above problems. My other objection to the use of an exception hierarchy is that I feel that inheritance is primarily a way of coding variations in behaviour. In this case, however, the variation isn't in the exceptions; it's in the handler. Therefore normal control structures in the handler should suffice and having a single exception class reduces the coupling in the application.

Some sources – particularly in the Java camp – suggest nesting exceptions. Again, I have found little use for this or for other advice such as the use of checked exceptions in Java. The simplicity of the proposed approach is appealing and has served me well on small programs. Extending this approach unmodified to a larger scale is possible but unlikely. More likely is to place try/catch blocks at component boundaries and use return codes across the interface with exceptions internally. This allows each component to look after its own cleanup and resource management without imposing anything on the caller. Cross-component error reporting is not something that you want to tie into a component's control flow; the raising of events or logging of errors is easier, cleaner and more usual. Throwing exceptions across network or process boundaries isn't a clean idiom either, so good old error codes are best here too. Handling errors on a system-wide scale is a different matter altogether. Languages like Erlang lead the way here.

This style of design relies on the use of exception safe techniques throughout the code. The strong guarantee – which offers commit/rollback semantics – should be used wherever it is practicable. Intermediate code should be exception neutral (i.e. have no try/catch blocks for cleanup) as this makes code clearer and easier to test as there is then no difference between the normal path and the path taken when an exception is thrown from lower-level code. These techniques have been well documented by members of the C++ community: auto_ptr, RAII, copy-swap idiom, "do work on the side", etc [Sutter99], [Stroustrup00]. One point to note is that the transactional nature of the strong guarantee is often provided for persistent data by using a relational database. For file storage, no such transactionality is available and developers must code it themselves. The same goes for in-memory state changes (although software transactional memory is a current research topic [STM05]). Rollback in such cases is provided not at the recovery point (the catch handler) but by the stack unwinding caused by the propagation of the exception. Designing commit mechanisms that do not throw can itself be a challenge!

Things that don't work and anti-patterns

I have seen a number of exception anti-patterns in C++ code in addition to the multiple subclasses anti-pattern:

- Try/catch surrounding each call
- Exception squashing (no recovery)
- Overuse of exceptions
- Throws in catch blocks

One anti-pattern I have often seen is surrounding each call with a try/catch block in order to add error context for logging:

```
void FunctionX()
```

```
{
      try {
            FunctionY();
      } catch (const MyException & e) {
            Log(e);
            throw MyException("Error: FunctionX -> FunctionY", e);
      }
      // similarly for call to FunctionZ()
}
```

This is worse than error codes for a number of reasons. The clarity of exception neutrality is lost, so the main application code is intertwined with error handling in exactly the way return codes are. When justifying this approach, programmers have told me that they believe that the extra context information is useful for debugging. I have never found it so as the most relevant information is available at the point where the error is detected, that is, where the exception is thrown. Adding in all of these intermediate try/catches is also a heavy burden that bloats the code and adds no user-visible functionality or tackles the real purpose of exception: recovery. I presume that developers who do this do not understand the purpose of stack unwinding and are still thinking in a C-style idiom.

If this approach is avoided, then the next item up the anti-pattern food chain is the null recovery block. This is where the catch handler either ignores the exception totally:

```
try {
      //
} catch (const MyException &) {}
```

because the programmer doesn't know what to do (and the specification is silent) or attempts to stumble on regardless. This anti-pattern is more common in Java because of the presence of checked exceptions as it can eliminate the need to alter the exception specification of the function. (InterruptedException springs to mind here.) There is little need to say why this is not a recommended practice! Just logging the error and continuing is almost as bad. (The "last gasp" handler falls into this category but since it by definition is called only when recovery is not possible then we turn a blind eye in this case.)

Some people overuse exceptions and use them when local control structures would be more appropriate. An example might be using an end-of-file exception instead of an error code, an event that is entirely predictable and something that does not need long-range recovery.

The final anti-pattern – throwing from within a catch block – is rarer and is more of a sin of omission than one of poor design or limited understanding. Sometimes this is exactly what is required when recovery at an intermediate level has failed or isn't possible, but often it is caused by calling a function that itself throws. Caveat coder. Examples abound in Java of calling jdbcStatement.close() and friends in finally blocks with a null object reference. The use of exception neutral intermediates reduces the number of places that this anti-pattern can occur to a few recovery points where additional thought, care and review time can be expended.

Consequences
Viewing exceptions as recovery requests has a number of important consequences. The primary effect is that it alters the perspective of the developer from a historic "stuff happens" view to a forward-looking "so what am I going to do about it" view, a change of focus that opens up a lot of possibilities as well as posing a large number of requirements-level questions about what recovery means. In "agile" terminology it is the start of a conversation with the customer that may well lead to some additional user stories. In a classic RUP-style use-case based approach it starts to fill in the "errors, exception and alternative paths" section of a full-dress Cockburn-style use case [Cockburn00]. This change in viewpoint should not be underestimated as it turns exceptional conditions from technical "oops" moments into user-visible events, events that contribute to the robustness and resilience of the application and about which users will usually have strong opinions. Altering a "can't save file foo.doc" show-stopping modal popup into a sequence of alternative actions – such as saving under a different name, on a separate device, etc – transforms an application and users' trust in it. The change of emphasis could be compared to the change that occurs when developers use test-driven development: you see things from another angle, one that illuminates the specification more than the implementation.

Once recovery from an exception becomes a user-visible event then it becomes a target for testing. Traditionally, error handlers have been poorly tested as it is often extremely difficult to provoke suitable errors. Exception neutrality now shines brightly for two reasons: reduction of error paths and ease of testing. Since exception-neutral code has no explicit error paths then any path through the code – normal or exceptional – provides adequate coverage. Thus, error handling does not add to the McCabe complexity of the intermediates and only the recovery points need testing for exceptional cases. There are only a handful of such catch blocks in the system so only these few places that contain complex recovery code have to be tested independently. Doing this by using mock objects that always throw often means inserting abstract base classes or interfaces as substitution points for mock objects. The overall testing burden for error handling is thus lower, it is less invasive and the orphan child Recovery need no longer feel like Cinderella on a bad night. (One point to remember is that exception neutrality and exception safety are orthogonal issues. The lack of catch blocks or calls to uncaught_exception() does not imply that throwing an exception will not lead to subtle errors or resource leaks!)

Why don't programmers do this currently?
Given the glowing overview and snake-oil claims for this approach, why do programmers not use it? Habit and old idioms, low-level detail and suspicion of exceptions are some of the reasons. Lots of developers are quite conservative creatures of habit and stick to what they know. Others are attracted to the Lorelei call of the new. Only a few, or so it seems to me, take the time to analyse how they could improve the use of their current languages and tools.

Habit and old idioms Developers have been using return codes for years and they are comfortable with them. Return codes are simple and fit well with local control structures. I am not advocating abandoning return codes as they should be the first port of call when handling errors. Exceptions are useful as an escalation request only when all local attempts have failed, in a similar manner to avoiding bothering your boss until you've tried everything at your disposal and need outside help.

Low-level detail By this I mean being submerged in lots of low-level detail and not being able to see the larger design-level decisions. This is why I advocate determining the recovery points early on in a design as it brings complexity-relieving structure to an otherwise bald and unconvincing area of programming. Knowing that help is only a throw away does much to alleviate the angst of error handling.

Suspicion of exceptions This is a definite issue with some developers, primarily those who are very concerned about performance and memory size. In order to reason about this we need to examine a little how exception handling is typically implemented and its associated costs. There are two common implementation techniques: stack-based and table-based. The table-based approach incurs zero execution-time cost if an exception is not thrown but relies on a potentially large static table of locations. This table is used to determine which destructors need to be called based on the program counter when the exception is thrown. In memory-constrained environments this table may be unacceptable. In comparison a typical stack-based approach has a number of parts to it. Catch handlers are required in the function where the catches are declared. Unwind handlers are required in every function that allocates an object on the stack that has a non-trivial destructor. This may seem heavy, but compare this to doing the whole thing manually. You have to write the error propagation mechanism yourself using return codes, if statements, early returns, etc. The overall amount of code in the executable is probably similar and the efficiency isn't that different either - some stacking of objects versus lots of return code creation and checking. So the run-time efficiency (both speed and size) isn't that different, I suspect. Breaking the error process into stages, there are three separate parts: detection, propagation and recovery. The speed and size of detection is the same for both exceptions and return codes, as is recovery. It is only propagation that differs and this is typically the smallest cost of the three parts. Exceptions provide a built-in mechanism for this whereas with return codes you have to write it all yourself every time afresh. As with any other cut-and-paste type of code duplication, you now own that mechanism so you have to test it, maintain it, and so on for the lifetime of the code. Faced with this, a standard clean compiler-provided mechanism seems like a good deal to me.

There are other forms of efficiency to consider, however. With return codes there is always the chance that you will get it wrong by omitting to test a return code, plus the fact that your application code is now intimately entwined with error handling code which makes it more difficult to understand and get right. On top of this, consider how difficult it is to test your manual error handling code: you have to falsify all of the return codes for all of the possible paths. This leads me to think that in terms of programmer efficiency that exceptions win hands down.

Exceptions and error handling are afterthoughts
I often get the feeling that error handling is but an afterthought, something that gets smeared on afterwards to bring an application back to a superficially acceptable level of stability. I hope that encouraging developers to think of recovery rather than termination by providing an overall error-handling structure might encourage developers to avoid concentrating on only the “happy day” scenario. Perhaps one day their bosses might even give them time to do so too.

Exceptions guarantees and design

The modern C++ community has adopted Dave Abrahams' three exception guarantees: basic, strong and nothrow [Sutter99]. It is instructive to see how these essentially technical-level guarantees relate to higher-level design techniques such as statecharts. The basic guarantee ensures that an object is in a usable state that satisfies its invariant after an operation if an exception is thrown. In terms of a UML statechart this means that if an exception is thrown then that particular state machine instance can reappear in any of the states! The strong guarantee implies that either a transition occurs or the state machine stays in the original state (commit/rollback semantics), and the nothrow guarantee implies that the transition will occur. The lack of precision of the basic guarantee is one of the primary reasons for aiming for the strong guarantee whenever possible. For instance, in an e-commerce application that implements only the basic guarantee, an exception could empty your persistent shopping cart and log you off. Even worse it could mark your order as complete! Caveat guarantor.

If exceptions are used as user-visible recovery requests then the recovery strategy can be designed as part of the same state machine as the normal path. This may involve additional operations, compensating transactions for rollback (as in two-phase commit), timeouts, etc. This reduces the likelihood of error handling being left up to developers who will probably choose the easiest option for them in the absence of clear requirements. My personal hope is that software designed this way will be far less likely to present me with a message box containing hexadecimal information barely helpful even to a developer, leading to reasonable software that isn't brittle, software that I can trust.

An interesting side issue is the use of assertions in debug mode versus production mode. The standard assert macro stops the program when in debug mode but not in the release version. Tony Hoare [Hoare73] commented that removing assertions in release mode is like wearing your lifebelt when practicing and removing it when venturing out for real. There would appear to be a case for having run-time assertions raise exceptions instead and then relying on the recovery mechanism to return the program to a usable state in both modes. This path seems to steer a fine line between assert's "stop the world" and "forget about assertions" modes, neither of which seems useful in production software.

Conclusion

Using exceptions as recovery requests is a technique I have been using successfully for a number of years in C++. The overall effect on the design is that the mainline code is simpler and clearer than with return codes and the exception handling provides a simple and stable structure that supports the mainline code by removing and isolating complexity. Sophisticated recovery strategies can be implemented, ones that involve a dialogue between the recovery code and the underlying functions.

This approach is very simple, seeming almost trivial compared to some of the anti-pattern approaches I have had the privilege to witness. I believe it expresses the essence of the approach in a clear manner, one that steers developers away from a historical technical viewpoint to a forward-looking user-oriented view of error handling. Perhaps this is its greatest strength, something only obvious in hindsight and something that grows on you over time.

Acknowledgements
The reviewers, Ric Parkin and Roger Orr, pointed out not only where I stepped off the narrow path of fact and into the realms of rose-tinted memory but also suggested a number of useful clarifications and alternatives. My thanks to them both.

References

[Sutter99] H. Sutter, *Exceptional C++,* Addison-Wesley, 1999.

[Stroustrup00] B. Stroustrup, *The C++ Programming Language (3rd Edition)*, Addison-Wesley, 2000.

[Hoare73] C.A.R. Hoare, *Hints on Programming Language Design*, Stanford University Artificial Intelligence memo AIM224/STAN-CS-73-403. Reprinted in [Hoare89], 193-214.

[Hoare89] C.A.R. Hoare/C.B. Jones (Eds.), *Essays in Computing Science* (reprints of Hoare's papers), Prentice Hall, 1989.

[STM06] http://research.microsoft.com/~simonpj/papers/stm/index.htm

[Cockburn00] A. Cockburn, *Writing Effective Use Cases*, Addison-Wesley, 2000.