IEEE Software
www.computer.org/software

Toward Exception-Handling Best Practices and Patterns

Rebecca J. Wirfs-Brock

Vol. 23, No. 5
September/October 2006

Editor: Rebecca J. Wirfs-Brock ■ Wirfs-Brock Associates ■ rebecca@wirfs-brock.com

Toward Exception-Handling Best Practices and Patterns

Rebecca J. Wirfs-Brock

I have long (but quietly) advocated dealing with exception handling issues early in the design of a system.

—John Goodenough, *Advances in Exception Handling Techniques*

Poor exception-handling implementations can thwart even the best design. It's high time we recognize exception handling's importance to an implementation's overall quality. I'm not comfortable relegating exception handling to a minor programming detail. Agreeing on a reasonable exception-handling style for your application and following a consistent set of exception-handling practices is crucial to implementing software that's easy to comprehend, evolve, and refactor. The longer you avoid exceptions, the harder it is to wedge cleanly designed exception-handling code into working software.

To demystify exception-handling design, we must write about—and more widely disseminate—proven techniques, guidelines, and patterns.

Exception basics

First, let's review the basics. An exception condition occurs when an object or component, for some reason, can't fulfill a responsibility. Or, in programming parlance, an exception condition occurs when some piece of code can't continue on its expected path. This could be owing to malformed arguments, an inconsistent state, bad data, unavailable resources, or coding or logic errors.

As a general rule, I don't explicitly design my software to detect and recover from logic or coding errors. It's better to find these errors using unit testing, treating the errors as bugs you must fix. But I do design my objects to explicitly verify argument values and check the availability of a shared resource. If the state of things isn't as expected, then an exception condition has been detected. Most popular programming languages explicitly support built-in exception mechanisms. In these languages, when you detect some condition that prevents your code from continuing on its normal execution path, you can signal an exception condition explicitly. For example, in Java and C#, you can create an exception object and write an expression to throw it. In addition to explicit exceptions that are raised by your code, the program execution environment can raise other exceptions, such as divide-by-zero errors or arrays indexed out of bounds. Regardless, at that point, your code can't continue on its normal path. Raising an exception breaks the normal execution flow.

When an exception is raised, the call stack unwinds until exception-handling code is encountered (in Java and C#, this is a catch block that identifies what class of exception it will handle). You're free to design your own exception classes or use preexisting exception

0740-7459/06/$20.00 © 2006 IEEE

classes. Your newly designed exception classes fit into and extend the existing exception class hierarchy.

By handling an exception and performing some sensible recovery action, you can put your software back into a known, predictable state. An alternative to raising an exception is to signal an exception condition via a return code or result object. In this case, instead of catching exceptions, your code either tests a return code and branches based on the value or queries the result object and makes decisions accordingly. A result object, just like an instance of an exception class, can also include extra information.

Exception-handling guidelines

Let's review some basic exception design guidelines, summarized from *Object Design: Roles, Responsibilities, and Collaborations* (Rebecca Wirfs-Brock and Alan McKean, Addison-Wesley, 2003).

Don't try to handle coding errors. Code that's designed for highly fault-tolerant systems might go to extraordinary efforts to detect and recover from exceptions. But unless your software is required to take such extraordinary measures, don't spend a lot of time designing it to detect and recover from programming errors. In the case of an out-of-bounds array index, divide-by-zero error, or any other programming error, the best strategy is to fail fast (and leave an audit trail of the problem that can be used to troubleshoot it).

Avoid declaring lots of exception classes. Create a new exception class when you expect it to be handled differently. Outside the world of exceptions, you wouldn't normally create two distinct classes to simply represent two different state values, so why create multiple exception classes? Create a new exception class when you expect some handling of the code to take a significantly different action, based on the exception type.

Name an exception after what went wrong, not who raised it. This makes it easier to associate the situation with the required action. If you name an exception after who threw it, it becomes less clear why the handler is performing the specific action.

Recast lower-level exceptions to higher-level ones whenever you raise an abstraction level. Don't let implementation details leak out of a method invocation as exceptions. Otherwise, your users might think your software is broken. When low-level exceptions percolate up to a high-level handler, there's little context to assist the handler in making informed decisions or reporting conditions that are traceable to any obvious cause. Recasting an exception whenever you cross an abstraction boundary enables exception handlers higher up in the call chain to make more informed decisions. If you want to include a problem trace when recasting them, you can always create a chained exception. A chained exception provides added context and holds a reference to the original lower-level exception. You can repeatedly chain exceptions.

I know of an application that reported a low-level I/O error and quit when it ran out of file space. The users thought the application had a bug. In fact, it had encountered an unrecoverable exception—there wasn't enough file space to continue. To correct customers' misimpressions and eliminate service calls, in a later release, the designers recast the low-level exception to a more meaningful one before passing it along. Customers still occasionally ran out of space (when the application created enormous temp files), but as a result of this slight coding change, they got an informative error message and knew to free up disk space before trying again.

Provide context along with an exception. What's most important in exception handling is information that helps create an informed response. Exception classes hold information. You can design them to be packed with information in addition to the bare-bones stack trace information provided by default. You might include values of parameters that raised the exception, specific error text, or detailed information that could be useful to plan a recovery.

Handle exceptions as close to the problem as you can. As a first line of defense, consider the initial requestor. If the caller knows enough to perform a corrective action, you can rectify the condition on the spot. If you propagate an exception far away from the source, it can be difficult to trace the source. Often objects further away from the problem can't make meaningful decisions.

Assign exception-handling responsibilities to objects that can make decisions. Although this seems to contradict the previous guideline, it doesn't. Often, the object best equipped to make a decision is the immediate caller. The caller asks this object to do X and it can't, but maybe it can jigger something and retry (see "Designing for Recovery," *IEEE Software,* July/August 2006, for a discussion of other recovery strategies). But sometimes the most able object is one that has been explicitly designed to make decisions and control the action. Controllers are naturals for handling exceptions as well as directing the normal flow of events.

Use exceptions only to signal emergencies. Exceptions shouldn't be raised to indicate normal branching conditions that will alter the flow in the calling code. For example, a find operation may return zero, one, or many objects, so I wouldn't raise an exception in this case. Instead, I'd design my `find()` method to return a null object or an empty collection. A dropped database connection, on the other hand, is a real emergency. There's nothing that can be done to continue as planned.

Don't repeatedly rethrow the same exception. Although exceptions don't cost anything until they're raised, programs that frequently raise exceptions run more slowly. A designer of a Web-based transactional system recounted how his team's convention of catching each exception, logging it, and then repeatedly rethrowing that same exception until some object in the call chain actually handled it slowed their application by 10 percent. An excess of fidgety catch-and-rethrow code can also confound efforts to track down a problem's source.

Patterns in progress

In writing this column, I researched other sources of exception-handling advice. The most wide-ranging discussion

I found online was at the Portland Pattern Repository (http://c2.com/cgi/wiki?ExceptionPatterns). Pattern descriptions are works in progress that can be freely extended and discussed by anyone who cares to chip in with knowledge, experience, examples, or counter-examples. Exception advice is currently organized into these pattern categories: defining exception types, raising exceptions, handling exceptions, exceptions and testing, when to use exceptions, alternatives to using exceptions, C++ idioms, C idioms, Java idiom, and discussion (a catchall category).

The "defining exception types" category includes Name the Problem not the Thrower, Exception per Context, Refine Exceptions, Homogenize Exceptions, and Generalize on Exception Behavior. For readability, I unrolled the jammed-together Portland Pattern Repository wiki name into a word phrase with spaces. So expect to see a slightly different form of the online name (the first exception pattern's wiki page is actually titled "NameTheProblemNotTheThrower").

Let me present one pattern to give you a taste. In the discussion of Name the Problem not the Thrower, `java.lang.ClassNotFoundException` is cited as an example of a well-named exception. This exception is thrown when an application tries to load it in a class that can't be found. Personally, I've found isolating the source of the problem to still be difficult, even if the exception is well- named (because class path problems can be confounding). Interestingly, `java.lang.NoSuchMethodException`, which at first blush seems reasonable, is cited as an example of a poorly named class. One place where this exception is raised is in the JavaBeans framework if a programmer fails to include an empty constructor for a bean class. I think this exception is aptly named. However, the exception class is poorly designed. Exception class designers can always choose to pack exception objects with facts useful in deciphering, troubleshooting, or recovering from an exception. In this case, the exception-class designer failed to include the missing method name.

From patterns in progress to full-blown exception patterns

Pattern descriptions in the Portland Pattern Repository range from terse to chatty. Because pattern hatching in this venue is a community process, advice accretes when someone takes interest and chips in. A pattern-in-progress reader must be patient and accepting of incomplete discussions and not fully validated ideas. It reminds me of standing around the proverbial water cooler with fellow developers, chatting about what works and why.

Just as important are discussions of what not to do, so-called *antipatterns*. For a brief discussion of some exception practices to avoid, look at Tim McCune's article, "Exception-Handling Antipatterns" (see http://today.java.net/pub/a/today/2006/04/06/exception-handling-antipatterns.html). Two of my favorites (having been repeatedly frustrated by them) are the Catch and Return Null and Throwing Exception antipatterns. The first is a sneaky way of obfuscating an exception condition. The second is a lazy way of declaring that a method might throw one or more exceptions without being explicit about which one (Java programmers are forced to declare all checked exceptions that are thrown in a method's signature). Another antipattern mentioned was Log and Throw. It seems more appropriate to me that Repeatedly Log and Rethrow is a better name for this antipattern. The first time you detect an exception, there might be a valid reason to log it to identify the point source. Repeatedly logging and rethrowing just adds clutter.

In truth, much work is needed to turn various sources of exception-handling advice and proto-patterns into an exception pattern design handbook. *Design Patterns* (Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, Addison-Wesley, 1995) established a high bar for design patterns, and there is no reason to lower the standard in the case of exceptions.

First and foremost, a pattern must be well-established. Gamma, Helm, Johnson, and Vlissides considered a design pattern only if it had been applied more than once in different systems. And although many other pattern authors have written about patterns less formally, these authors stated that they view the essential ingredients of a pattern to include its name, when to apply it, a general description of the solution, and the consequences—the results and tradeoffs a designer must make when applying the pattern. Consequences in their estimation could include impact on a system's flexibility, programming language-specific issues, or time and space considerations.

This basic structure and criteria seem quite appropriate for exception-handling patterns. The time is ripe for consolidating exception-handling knowledge into a set of polished best practices and patterns.

Rebecca J. Wirfs-Brock is president of Wirfs-Brock Associates and an adjunct professor at Oregon Health & Science University. She's also a board member of the Agile Alliance. Contact her at rebecca@wirfs-brock.com.