IEEE Software
www.computer.org/software

Designing for Recovery

Rebecca Wirfs-Brock

Vol. 23, No. 4
July/August 2006

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

Designing for Recovery

Rebecca Wirfs-Brock

The consequences of structural failure in nuclear plants are so great that extraordinary redundancies and large safety margins are incorporated into the designs. At the other extreme, the frailty of such disposable structures as shoelaces and light bulbs, whose failure is of little consequences, is accepted as a reasonable trade-off for an inexpensive product. For most in-between parts or structures, the choices are not so obvious.

—*Henry Petroski,* To Engineer is Human

When things go right, software hums along like well-oiled machinery—receive an event, twiddle with inputs, send a flurry of messages, change the system state, interact with the environment or users, then wait for the next chunk of work. Smooth. Mechanical. Predictable. But what happens when something goes wrong? How should you design your software to detect, react, and recover from exceptional conditions?

If you follow Jim Shore's advice and design with a fail fast attitude, you won't expend any effort recovering from failures ("Fail Fast," *IEEE Software*, Sept./Oct. 2004). Shore argues that a "patch up and proceed" strategy often obfuscates problems. For example, choosing to plug in a default value when your software can't find a parameter setting in a configuration file can lead your software to fail later or work unpredictably. Returning a null value isn't arguably better. What's the caller supposed to do with a null—set a default, report an error? That seems like pushing the problem away from its origin to a more uninformed source. Shore's simple design solution is to write code that checks for expected values upon entry and returns failure notifications when it can't fulfill its responsibilities. He argues that careful use of assertions allows for early and visible failure, so you can quickly identify and correct problems.

Failing fast is a reasonable option for dealing with programmer errors, but at times your software encounters rare but anticipated exception conditions. If these exceptional cases aren't handled well, users might view your software as buggy, riddled with logic errors. Most software needs to keep working in spite of many anticipated glitches. How gracefully it reacts and responds to anticipated exceptional cases is a measure of its quality.

Recovery options

Many scenarios exist in which failing fast wouldn't be your first choice. Consider when

- data is inconsistent but you still might be able to make sense of it;
- a query doesn't return expected results;
- a requested action isn't possible due to the software's current state, but conditions could change;
- a connection is intermittent;
- a resource isn't currently available;
- an external system doesn't respond as expected; or
- your software's view of the state of world doesn't match the views of other systems or your users.

0740-7459/06/$20.00 © 2006 IEEE

As designers, we have a choice in how much effort to expend designing recovery actions. To intelligently sort through the costs and appropriateness of various recovery scenarios and actions, it's useful to know your options. When something can't proceed as expected in a block of code, you can silently ignore the request, admit failure, resign, retry, perform an alternative action, or appeal to a higher authority.

Ignore the request

No, I didn't put "ignore the request" on the list just so I could quickly dismiss it. Most of the time, code that can't perform a request shouldn't swallow an exception and pretend that everything is okay. But when a failure to act is of no consequence to the requestor or overall system, this might not only be appropriate—it might be the only reasonable option.

Consider an interrupt handler that by design drops an event when it can't post it to a shared buffer. An animation might "jump" ahead in sequence or the latest stock quote might not get posted. Although not desirable, this might be the best you can do. Make this choice in good conscience when there aren't any serious consequences and when it won't compromise requirements.

Failing, really failing: Two options

Admitting failure is another simple option. As already discussed, if a method or function can't perform a request, you can design it to signal an exception, return an error condition, set some failure status indication, or log the failure (or a combination of notification and logging). Most modern programming languages provide an exception mechanism that can be used to signal failures. I prefer to raise exceptions only in the case of an emergency. Some extraordinary, atypical condition has caused my code to inextricably veer away from any reasonable course of action, and all I can do is signal that there's a problem. An example would be a database connection becoming unavailable in the middle of a query method.

However, no items matching a query isn't a failure on my code's part—just a normal, expected case. Queries don't always yield results. So, in that case, I wouldn't signal failure but would instead design my method to return an empty collection, null object, or some result that the caller could easily interpret to signify "nothing found but carry on."

Which brings me to your next option. If your software can't recover from a failed action, you can design it to resign—to fail a little more gracefully than failing fast. You might free up acquired resources, clean things up, and then signal *definite failure*. In this case, the caller need not attempt any further recovery because it isn't possible. By convention, declaring an unchecked exception in Java indicates that it's probably not easy to recover from this exception. But there's no definite way of signaling "I resign." Other programming languages offer no better option, so mechanisms for signaling resignation are left as implementation concerns.

Of course, when a method reports a failure, as opposed to a definite failure, recovery might be possible. Usually, the best choice is the caller or a specially designed recovery handler, which have enough contextual information to carry out the recovery. For example, if a query fails because a database connection is unavailable, it might make sense to attempt to reestablish that connection and query again. This leads me to the next recovery strategy.

I find arbitrary retry strategies extremely annoying. Is it really better to blithely retry three times rather than five, seven, or 23 times?

Intelligent retry

Retrying is sensible when the failure was likely caused by a temporary condition that you can either try to rectify through your own code or that's likely to change owing to actions outside your software's control. An external system might be momentarily busy or a shared resource temporarily unavailable. Sooner or later, the system will be freed up or the resource will become available.

But how much retrying should you do? As a designer, I find arbitrary retry strategies extremely annoying. Is it really better to blithely retry three times (which seems to be culturally acceptable but still nonsensical) than five, seven, or 23 times? It's much more satisfying to base retry strategies on reasoning, logic, and a deeper understanding of how the design will interact with the complex state of other systems, the network, or the hardware. I want retrying to actually increase my odds for success. But determining satisfactory, reasoned, and sound retry strategies can be difficult.

I recently spoke with a designer whose software had often failed to connect to a critical network resource. As initially designed, the software would fail after retrying a few times. Yet because this was part of a critical business transaction, the users would repeatedly attempt the transaction. Even more confounding was that they had to rekey some information before retrying the transaction. So, the designer explained the situation to his customers and asked how long the software should try to acquire the resource. The customers said one hour. Implementing this change resolved the issue, because the software almost always established a connection within an hour.

Involving customers and users

When failure isn't an option and retrying doesn't make sense, it might be appropriate to design an acceptable alternative. These aren't perfect—acceptable alternatives almost always represent compromises. And most of the time you must discuss, think through, and work out what's acceptable with several interested parties. If the software can't record critical information, even after we've retried, is it acceptable

to record it at a secondary location? If that secondary location doesn't work, should we invest in making incremental backups of that information as the software is running (at the risk of slowing things down)? If information is only partially complete, is it acceptable to interpret it in a particular way rather than reject it outright?

There are often no cut-and-dried answers. As designers, we should be prepared to offer alternatives and open up a dialog with our customers. Each choice we make has an associated development cost and might not even prove acceptable until it has been field tested.

Appealing to a higher authority

But software can't recover from every failure on its own. In certain situations, it might be appropriate to ask a human to apply judgment and steer the software to an acceptable resolution.

For several years, I worked on a complex telecommunications integration system that received product orders from external applications. It translated orders into work tasks, then sent requests to other systems to provision equipment, establish billing information, and complete the order. Sometimes requests to external systems would fail after all our recovery actions had been exhausted. When this happened, we designed our software to place an order on a problem queue and report it to a customer service representative. Admittedly, this didn't seem perfect, but we couldn't fix the problem without involving an actual person. Usually our software could correct things and proceed, but in this case, the best design choice was to report failure and appeal to a higher authority.

At the end of the day, designers must make intelligent choices on which recovery actions to take and when to give up. Not all recovery decisions should be left to the designer's discretion. David Pye, writing for industrial designers and building architects in *The Nature and Aesthetics of Design* (Cambien Press, 1995), cautions

> *The designer or his client has to choose to what degree and where there shall be failure. Thus the shape of all designed things is the product of arbitrary choice. … It is quite impossible for any design to be the "logical outcome of the requirements" simply because requirements being in conflict, their logical outcome is an impossibility.*

Pye's words ring true for software designs as well. Hopefully, we don't make tough design decisions in a vacuum. In an ideal world, determining how to respond to failure (and how hard to try to recover) should be a joint decision made by both the software designers and their informed customers—not just by one or the other.

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.