Wednesday, March 14, 2012

Defensive Programming: Being Just-Enough Paranoid

> Hey, let's be careful out there.
> Sergeant Esterhaus, daily briefing to the force of Hill Street Blues

When developers run into an unexpected bug and can't fix it, they'll "add some defensive code" to make the code safer and to make it easier to find the problem. Sometimes just doing this will make the problem go away. They'll tighten up data validation – making sure to check input and output fields and return values. Review and improve error handling – maybe add some checking around "impossible" conditions. Add some helpful logging and diagnostics. In other words, the kind of code that should have been there in the first place.

Expect the Unexpected

> The whole point of defensive programming is guarding against errors you don't expect.
> Steve McConnell, Code Complete

The few basic rules of defensive programming are explained in a short chapter in Steve McConnell's classic book on programming, Code Complete:

1. Protect your code from invalid data coming from "outside", wherever you decide "outside" is. Data from an external system or the user or a file, or any data from outside of the module/component. Establish "barricades" or "safe zones" or "trust boundaries" – everything outside of the boundary is dangerous, everything inside of the boundary is safe. In the barricade code, validate all input data: check all input parameters for the correct type, length, and range of values. Double check for limits and bounds.
2. After you have checked for bad data, decide how to handle it. **Defensive Programming is NOT about swallowing errors or hiding bugs.** It's about deciding on the trade-off between robustness (keep running if there is a problem you can deal with) and correctness (never return inaccurate results). Choose a strategy to deal with bad data: return an error and stop right away (fast fail), return a neutral value, substitute data values, … Make sure that the strategy is clear and consistent.
3. Don't assume that a function call or method call outside of your code will work as advertised. Make sure that you understand and test error handling around external APIs and libraries.
4. Use assertions to document assumptions and to highlight "impossible" conditions, at least in development and testing. This is especially important in large systems that have been maintained by different people over time, or in high-reliability code.
5. Add diagnostic code, logging and tracing intelligently to help explain what's going on at run-time, especially if you run into a problem.
6. Standardize error handling. Decide how to handle "normal errors" or "expected errors" and warnings, and do all of this consistently.
7. Use exception handling only when you need to, and make sure that you understand the language's exception handler inside out.

> Programs that use exceptions as part of their normal processing suffer from all the readability and maintainability problems of classic spaghetti code.
> The Pragmatic Programmer

I would add a couple of other rules. From Michael Nygard's Release It! never ever wait forever on an external call, especially a remote call. Forever can be a long time when something goes wrong. Use time-out/retry logic and his Circuit Breaker stability pattern to deal with remote failures.

And for languages like C and C++, defensive programming also includes using safe function calls to avoid buffer overflows and common coding mistakes.

Different Kinds of Paranoia

The Pragmatic Programmer describes defensive programming as "Pragmatic Paranoia". Protect your code from other people's mistakes, and your own mistakes. If in doubt, validate. Check for data consistency and integrity. You can't test for every error, so use assertions and exception handlers for things that "can't happen". Learn from failures in test and production – if this failed, look for what else can fail. Focus on critical sections of code – the core, the code that runs the business.

Healthy Paranoid Programming is the right kind of programming. But paranoia can be taken too far. In the Error Handling chapter of Clean Code, Michael Feathers cautions that

> "many code bases are dominated by error handling"

Too much error handling code not only obscures the main path of the code (what the code is actually trying to do), but it also obscures the error handling logic itself – so that it is harder to get it right, harder to review and test, and harder to change without making mistakes. Instead of making the code more resilient and safer, it can actually make the code more error-prone and brittle.

There's healthy paranoia, then there's over-the-top-error-checking, and then there's bat shit crazy crippling paranoia – where defensive programming takes over and turns in on itself.

The first real world system I worked on was a "Store and Forward" network control system for servers (they were called minicomputers back then) across the US and Canada. It shared data between distributed systems, scheduled jobs, and coordinated reporting across the network. It was designed to

The first real world system I worked on was a “Store and Forward” network control system for servers (they were called minicomputers back then) across the US and Canada. It shared data between distributed systems, scheduled jobs, and coordinated reporting across the network. It was designed to be resilient to network problems and automatically recover and restart from operational failures. This was ground breaking stuff at the time, and a hell of a technical challenge.

The original programmer on this system didn’t trust the network, didn’t trust the O/S, didn’t trust Operations, didn’t trust other people’s code, and didn’t trust his own code – for good reason. He was a chemical engineer turned self-taught system programmer who drank a lot while coding late at night and wrote thousands of lines of unstructured FORTRAN and Assembler under the influence. The code was full of error checking and self diagnostics and error-correcting code, the files and data packets had their own checksums and file-level passwords and hidden control labels, and there was lots of code to handle sequence accounting exceptions and timing-related problems – code that mostly worked most of the time. If something went wrong that it couldn’t recover from, programs would crash and report a “label of exit” and dump the contents of variables – like today’s stack traces. You could theoretically use this information to walk back through the code to figure out what the hell happened. None of this looked anything like anything that I learned about in school. Reading and working with this code was like programming your way out of Arkham Asylum.

If the programmer ran into bugs and couldn’t fix them, that wouldn’t stop him. He would find a way to work around the bugs and make the system keep running. Then later after he left the company, I would find and fix a bug and congratulate myself until it broke some “error-correcting” code somewhere else in the network that now depended on the bug being there. So after I finally figured out what was going on, I took out as much of this “protection” as I could safely remove, and cleaned up the error handling so that I could actually maintain the system without losing what was left of my mind. I setup trust boundaries for the code – although I didn’t know that’s what it was called then – deciding what data couldn’t be trusted and what could. Once this was done I was able to simplify the defensive code so that I could make changes without the system falling over itself, and still protect the core code from bad data, mistakes in the rest of the code, and operational problems.

Making code safer is simple

The point of defensive coding is to make the code safer and to help whoever is going to maintain and support the code – not make their job harder. Defensive code is code – all code has bugs, and, because defensive code is dealing with exceptions, it is especially hard to test and to be sure that it will work when it has to. Understanding what conditions to check for and how much defensive coding is needed takes experience, working with code in production and seeing what can go wrong in the real world.

A lot of the work involved in designing and building secure, resilient systems is technically difficult or expensive. Defensive programming is neither – like defensive driving, it’s something that everyone can understand and do. It requires discipline and awareness and attention to detail, but it’s something that we all need to do if we want to make the world safe.

Posted by Jim Bird at