How to Build Reliable Code

How to Build Reliable Code

Oliver Sharp

The first thing to understand: It is hard to build complex software that works well. In the search for salvation, or what software engineer and author Fred Brooks calls the silver bullet, many people look to models, techniques, and tools. Once upon a time, the solutions were structured programming and high-level languages; now, they're applications builders, componentware, and object-oriented-programming (OOP) techniques. However, evangelists for all these solutions ignore an uncomfortable truth: Reliable software can be written using gotos and assembly language, and truly dismal code has been produced using impeccably modern tools and techniques.

The reality is that one factor completely dominates every other in determining software quality: how well the project is managed. The development team must know what code it is supposed to build, must test the software constantly as it evolves, and must be willing to sacrifice some development speed on the altar of reliability. The leaders of the team need to establish a policy for how code is built and tested. Tools are valuable because they make it easier to *implement* a policy, but they can't define it. That is the job of the team leaders, and if they fail to do it, no tool or technique will save them.

One reason that quality often takes a backseat is that it is not free. Reliable software often has fewer features and takes longer to produce. No trick or technique will eliminate the complexity of a modern application, but here are a few ideas that can help.

Fight for a Stable Design

One of the worst obstacles to building a good system is a design that keeps changing. Each change means redoing code that has already been written, shifting plans in midstream, and corrupting the internal consistency of the system.

The problem is that often nobody knows what the program should do until there is a preliminary version to run. An excellent strategy is to build mock-ups and prototypes that potential users can start working with early, so that the design settles down as soon as possible. Once designers hammer out the basic structure of the system, any changes that aren't critical should wait until the next version. This is a hard line to hold, but the closer developers can come to it, the better off the code will be.

Cleanly Divide Up Tasks

When designing a complex system, divide the work into smaller pieces that have good interfaces and share the appropriate data structures. If you get that right, you can make many bad implementation decisions without ruining the overall design and performance of the system.

Object-oriented languages can be a usef ul way to express and enforce the decomposition strategy, but they don't tell the designer how to do the job. It is infinitely better to have a good design implemented in C than a poor one in C++.

Avoid Shortcuts

Programmers often don't take time to fix a design error as the code evolves. Those decisions can come back to haunt everyone. Avoid shortcuts by insisting that each one is carefully documented. The pain of writing something up can act as a useful deterrent.

Use Assertions Liberally

An assertion is simply a line of code that says, "I think this is true. If it isn't, something is wrong, so stop execution and let me know right away." If a value is supposed to be within a certain range, check first. Make sure that pointers point somewhere and that internal data structures are consistent.

Just like other debugging code, you can compile assertions out of production code before it enters final testing stages. There is every reason to lit ter your code with assertions. You will find problems quickly, making them much easier to track down.

Use Tools Judiciously

Tools are not a panacea -- they can't help you fix a project that is being administered badly. But tools can make it easier for development teams to put good policies into effect. Source code management tools, such as the public domain RCS or PVCS from Intersolv, help you coordinate modules being used by multiple developers.

There are also some tools that can find certain errors in your code instead of forcing you to do it. The Unix utility `lint` (or the turbo-charged version offered in Centerline's Code Center) will find syntax errors and mismatches between different source code files. Purify, from Pure Software, and BoundsChecker, from Nu-Mega Technologies, catch a wide variety of memory errors when they occur, rather than when they manifest themselves later on. Other tools perform regression tests or do code-coverage analysis to see if th ere are dusty corners of your program that are not being exercised.

Rely on Fewer Programmers

An easy way to reduce the number of bugs in a project is to cut down on the number of people who are involved in it. The advantages are less management overhead, less need for coordination, and more contact among the team members who are building the system.

You can reduce the number of people by having individual programmers produce code more quickly or by reducing the amount of code that needs to be written. CASE tools, applications builders, and code reuse are all attempts to meet one or both of these goals. While these products don't always live up to their promise, they can simplify a project so that a smaller team can handle it.

---

9 Ways to Write More-Reliable Software

`--  `**`Fight`**` for a Stable Design`

-- **Cleanly** Divide Up Tasks

-- **Avoid** Shortcuts

-- **Use** Assertions Liberally

-- **Use** Tools Judiciously

-- **Rely** on Fewer Programmers

-- **Diligently** Fight Featuritus

-- **Use** Formal Methods Where Appropriate

-- **Begin** Testing Once You Write the First Line of Code

Oliver Sharp is the director of consulting services at Colusa Software (Berkeley, CA). You can contact him on the Internet at oliver.sharp@colusa.com .