acmqueue

Broken Builds

Frequent broken builds could be symptomatic of deeper problems within a development project.

Dear KV,

Is there anything more aggravating to programmers than fellow team members checking in code that breaks a build? I find myself constantly tracking down minor mistakes in other people's code simply because they didn't check that their changes didn't break the build. The worst part is when someone has broken the build and they get indignant about my pointing it out. Are there any better ways to protect against these types of problems?

Made to be Broken

Dear Made,

I know you, and everyone else, are expecting me simply to rant about how you should cut off the tips of the pinkies of the offending parties as a lesson to them and a warning to others about carelessness. While that might be satisfying, it's illegal in most places and, I'm told, morally wrong.

A frequently broken build is a symptom of a disease, but it is not the disease itself. It indicates problems in any of the following three areas: management, infrastructure, or software architecture.

Management is the area that most quickly comes to mind when there is a team- or project-wide problem. The belief of most of the workers on a project—those tasked with writing and verifying code and systems—is that project-wide problems need to be solved by Mommy (aka the project lead or the manager). Unfortunately, Mommy can remind people only so often to clean up their rooms, to tie their shoes, and not to check in broken code.

One of the best solutions to the problem of people not checking their code before they check it in is peer pressure. Anyone who checks in code without compiling it first ought to feel embarrassed by such a mistake, and if not, the other people around them should strongly encourage them to feel embarrassed. Shame, it turns out, is a strong motivator for avoiding antisocial behavior. Like many—or perhaps all—of KV's suggestions, shaming can be taken too far, but I suggest you try it and see how it works.

Depending on Mommy to tell off the misbehaving kids becomes tiresome both for you and the project management after a while. What you want to see is a good working culture develop, one in which people know that breaking the build is like taking a dump in the middle of the break room; funny once, but usually unacceptable.

Poor infrastructure can also lead to suffering with frequently broken builds. One thing that continues to amaze me is how computer hardware gets cheaper, and yet companies continue to coast along without a nightly, or more frequent, build system. For the price of a single desktop computer and a few days of scripting, most teams can have a system that periodically updates a test build of their code, builds it, and sends e-mail to the team if the build fails. The amount of time saved by such a system is easily measurable. Subtract 1 from the number of programmers on a team. Multiply the resulting number by the number of hours it usually takes to figure out who broke the build, find them, shame them, and have them fix the build. Now multiply THAT number by the average hourly

wage of each person on the team, and you have a rough idea of how much time and money was wasted by not having periodic builds. We won't get into periodic testing, which can save even more time and money, because if your build is always broken, you clearly have not achieved a sufficient level of sophistication to move on to nightly tests.

Even though the broken code will still get into the system, with a periodic build system the offending person will find out fairly quickly that he or she broke the build and hopefully will admit it in an e-mail ("I broke the build, hang on a second") and then repair the error. While this is still suboptimal, it is far better than what you had before.

Sometimes it is the build system itself that is the source of the problem. Many modern build systems depend heavily on caching derived objects, as well as the parallelization of the build process. While a parallel build process can provide you results more quickly, it can often lead to build failures that are false positives. Trying to build an object that requires another object to be created first, such as an automatically created include file, always leads to trouble. Maintaining the list of dependencies by hand is an error-prone, but often necessary, process. If you are using a build system that depends on caching and uses parallel builds, then your problems may lie here.

Now we come to the final area that is the cause of build problems. The way in which a piece of software is put together, frequently referred to as its architecture, often impacts not only how the software performs when it runs, but also how it is built. I hesitate to use the word *architecture* since overuse of the term has led to the unfortunate proliferation of the job title software architect, which is far too often a misnomer.

If all the components of a software system are too interdependent, then a change to one can result in an injury to all. A lack of sufficient modularization is often a problem when software ships, but it is definitely a problem when the software is being compiled. When a change to an include file in one area leads to the build breaking in another area, then your software is probably too heavily interlinked, and the team should look at breaking the pieces apart. Often such links come from careless reuse of some part of the system. Careless reuse is when you look at a large abstraction and think, "Oh, I really want this version of method X," where X is a small part of the overall abstraction, and then you wind up making your code depend not just on the small part you want, but on all of the parts that X is associated with. If you get to the point where you know that it's neither carelessness nor poor infrastructure that is leading to frequent build failures, then it's time to look at the software architecture.

Now you know the three most basic ways to alleviate frequent build breakage: shaming your teammates, adding some basic infrastructure, and finally improving the software architecture. That ought to keep you out of jail, for now.

KV

KODE VICIOUS, known to mere mortals as George V. Neville-Neil, works on networking and operating system code for fun and profit. He also teaches courses on various subjects related to programming. His areas of interest are code spelunking, operating systems, and rewriting your bad code (OK, maybe not that last one). He earned his bachelor's degree in computer science at Northeastern University in Boston, Massachusetts, and is a member of ACM, the Usenix Association, and IEEE. He is an avid bicyclist and traveler who currently lives in New York City.