acmqueue

Commitment Issues

When is the right time to commit changes?

Dear KV,

One of the other people on my project insists on checking in unrelated changes in large batches. When I say *unrelated*, what I mean is he will fix several unrelated bugs and then make a few minor changes to spacing and indentation across the entire source tree. He will then commit all of these changes at once, usually with a short commit message that lists only the bugs he claims to have fixed. Do you think I'm being too picky in wanting each checkin to address only one issue or problem?

Whited Out

Dear Whited,

In all the years I've been writing these pieces, I don't think I've ever told anyone they were being too picky—which, given how much I tend to complain and point out other people's faults, is a bit surprising.

Of course, if I were in your shoes I wouldn't just complain about it; I would back out the person's commit and make him commit the work in smaller chunks. Happily, modern source-code control systems cut both ways: they make it easy to group commits, and they make it easy to reverse them.

Not only do I prefer commits to have only related changes, but I also think there are distinct types of commits and that programmers should make sure their commits are not leaky. Leaky commits usually arise because of the "just one more thing" syndrome. You start out fixing a bug but then find that the bug is the result of code that is not only poorly structured but also poorly formatted. Once the bug is fixed, you naturally want to clean up the other gunk that you've uncovered during your fix, such as poor indentation, poor use of white space, lack of comments, and so on. Tidying up a piece of code is something to be praised, but rototilling a whole file while fixing a bug is not. As usual, it's a question of balance.

The right way to handle bit rot or ugly code that you find while working on something else is to stop, check in the fix, and then go rototill the file! Make sure to do the cleanup as soon as possible after your bug fix because we both know that something more pressing is quite likely to pop up as soon as you hit Return on the commit for your little bug fix. Suddenly an e-mail will arrive from your boss asking you to fix another bug, and then you won't have time to clean up the gunk you found earlier, which will lead to more bugs being found in it, and more internal pressure to actually "get it right, damn it!" and eventually you will wind up banging your head on office windows and yelling a lot. Well, maybe you won't, but this is what happens to me when the list of files that I know must be cleaned up, but that I don't have the time to clean up, grows too long.

One way to avoid being interrupted while rototilling a file is to group commits, which is what you ought to teach your coworker to do. It's fine to fix a few bugs, clean up some files, add a comment here, remove stray white space there, and do a good job of all of it. The key is to make sure that each of the changes can be committed independently. The best way of achieving this is to work in your

own branch, one in which you can commit without everyone else seeing your work immediately. Commit each change independently—first the bug fixes, then the broad rototilling, and finally the white-space and comment cleanup—into your own branch and then merge all the changes into the actual work tree.

This approach is not without risks—in particular, you’ll be stuck in merge hell (see Merge Early, Merge Often, KV, October 2009; <http://queue.acm.org/detail.cfm?id=1643030>) for some time—but if you’re doing bug fixes and white-space changes, you should be able to merge your work weekly at the very least, which should lower the amount of merging you have to do. If you can, merge the changes daily, which will help to contain the amount of merging you have to do.

A word about white-space and formatting changes. Although it is virtuous to make sure that all the code in a project is well formatted and adheres to the coding standard, it is a very, very antisocial act to reformat code just for the sake of reformatting it. Every line you change is going to show up in a diff between the versions, and it is incredibly maddening to try to compare two versions of a file across a massive white-space diff. Sometimes you just have to live with the spacing or formatting in a file in order to preserve the necessary history—and everyone’s sanity. Alas, this is another piece of advice that requires balance. I can’t tell you how much white-space change is too much, but I can tell you that if you change the formatting of a file and it changes every line, you had better be willing to defend yourself against a group of angry coworkers who may see your change as far too disruptive with little or no tangible benefit.

KV

Dear KV,

Do you ever embed tests into your code, as opposed to writing independent test code, as a simple way of testing basic functionality? This is how I was taught to test my code in school, but I rarely see it done in practice.

Along For the Ride

Dear Along,

Adding internal tests was not something I learned in school but is something that I’ve seen, and occasionally used, in practice. My two favorite examples of this type of testing are in calculator code and in the vi editor.

Once upon a time, many people, including KV, used powerful calculators that allowed you to program them. The programs were distributed on paper, often in magazines, and had to be keyed in on the calculator directly. This was an error-prone process. If you were very lucky or rich, you might be able to read the programs off of the page using a bar-code reader. Keying in the programs was laborious, but fixing them—by editing the program one line at a time—was even less fun. Often each subroutine had a comment saying something like, “Enter these parameters into the calculator and execute this subroutine; if the answer is not X, then you did something wrong; go back and fix the code.” In this way, as you were keying in the code, you could check, bit by bit, that what you had typed in was correct, thereby improving your chances of success with the whole program and probably also helping to lower your blood pressure and to live a longer and happier life.

Enough about ancient history; let’s move forward a bit. Many of you use the vi editor or more likely its newer cousins nvi and vim. I hope you’ve never had to read the internals of the code,

because it is very scary. Many years ago Keith Bostic reworked the vi editor into nvi and did a great job of making a very confusing program somewhat less confusing and somewhat more maintainable. I did a small bit of work extending the editor and began to come across nuggets such as these:

```
* !!!
* For the fun of it, if you want to see if a vi clone got the ex argument
* parsing right, try:
*
*     echo 'foo|bar' > file1; echo 'foo/bar' > file2;
*     vi
*     :edit +1|s|/PIPE/|w file1| e file2|1 | s|//SLASH/|wq
*
* or:  vi
*      :set|file|append|set|file
*
* For extra credit, try them in a startup.exrc file.
*
```

Sprinkled throughout the code were many of these embedded tests meant to help anyone reading the code know, relatively quickly, if they had made a mistake when they made a change.

These kinds of tests work relatively well when it is easy to see the output change. They would not work very well in, for example, a network stack, because you're likely to have to look at the contents of a packet. While working on a code editor, you could embed the test right into the source code, because it's very likely that you're using the very same editor to work on the code and can try out the suggested test and get the answer immediately.

One type of code that is amenable to these sorts of embedded tests is code that is in an interpreted language. Some program editors have a way of executing a region of code that's being edited as a way of testing just that subroutine. Editors for the Lisp programming language have long had this feature, and it has been extended in the Emacs editor to cover other scripting languages such as Python and Perl. If you are working on interpreted code with an editor that allows the code to be executed *in situ*, then you should definitely be adding these kinds of tests to your code.

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.

© 2010 ACM 1542-7730/10/0200 \$10.00