DOI:10.1145/2492007.2492019

George V. Neville-Neil

Article development led by acmqueue
queue.acm.org

Kode Vicious
Cherry-Picking and the Scientific Method

Software is supposed be a part of computer science, and science demands proof.

Dear KV,
I have spent the past three weeks trying to cherry-pick changes out of one branch into another. When do I just give up and merge?
In the Pits

Dear Pits,
I once rode home with a friend from a computer conference in Monterey. It just so happened that this friend is a huge fan of fresh cherries, and when he saw a small stand selling baskets of them he stopped to buy some. Another trait this friend possesses is that he can't ever pass up a good deal. So while haggling with the cherry seller, it became obvious that buying a whole flat of cherries would be a better deal than buying a single basket, even though that was all we really wanted. Not wanting to pass up a deal, however, my friend bought the entire flat and off we went—eating and talking. It took another 45 minutes to get home, and during that time we had eaten more than half the flat of cherries. I could not look at anything even remotely cherry-flavored for months; and today, when someone says "cherry-picking," that does not conjure up happy images of privileged kids playing farmer on Saturday mornings along the California coast—I just feel ill.

PHOTOGRAPH BY ARINA HABICH

All of which brings me to your letter. It is always difficult to say when someone else should "just give up and do X" no matter what X is, but at some point you know—deep down, somewhere in that place that makes you an engineer—what started out as a quick bit of cherry-picking has turned into a horrific slog through the mud, and nothing short of a John Deere tractor is going to get you out of it. The happy moments in the sunshine have ended, it is raining, you are cold, and you just want to go home. That is the time to stop and try again.

I know this probably ought to

go without saying, but the real reason most of us wind up in the pits of cherry-picking is because we have not been doing the real work of periodically merging whatever code we are working against. We have let the head of the tree, or the tip of the git, or whatever trite phrase people might want to use, get away from us, and the longer we wait to do the merge, the more pain we are going to suffer. The best way to keep from being stuck in the cherry orchard is to have a merged and tested branch ready to go when it is time for your project to resynchronize with the head of the development tree. I know this is more work than isolating yourself in a corner and just working on the next release, but in the end it will save you a lot of headaches. The question next time won't be, "When do I stop cherry-picking?" but simply, "When is the new branch ready to receive the work we have already done?"

KV

Dear KV,

I just started working for a new project lead who has an extremely annoying habit. Whenever I fix a bug and check in the fix to our code repository, she asks, "How do you know this is fixed?" or something like that, questioning every change I make to the system. It is as if she does not trust me to do my job. I always update our tests when I fix a bug, and that should be enough, don't you think? What does she want, a formal proof of correctness?

I Know Because I Know

Dear I Know,

Working on software is more than just knowing in your gut that the code is correct. In actuality, no part of working on software should be based on gut feelings, because, after all, software is supposed be a part of computer science, and science demands proof.

One of the problems I have with the current crop of bug-tracking systems—and trust me, this is only one of the problems I have with them—is that they do not do a good job tracking the work you have done to fix a bug. Most bug trackers have many states a bug can go through: new, open, analyzed, fixed, resolved, closed, and so forth, but that is only part of the story of fixing a bug, or doing anything else with a program of any size.

When you approach a problem, you should do it in a way that mirrors the scientific method.

A program is an expression of some sort of system that you, or a team, are implementing by writing it down as code. Because it is a system, one has to have some way of reasoning about that system. Many people will now leap up and yell, "Type Systems!", "Proofs!", and other things about which most working programmers have no idea and are not likely ever to come into contact with. There is, however, a simpler way of approaching this problem that does not depend on a fancy or esoteric programming language: use the scientific method.

When you approach a problem, you should do it in a way that mirrors the scientific method. You probably have an idea of what the problem is. Write that down as your theory. A theory explains some observable facts about the system. Based on your theory, you develop one or more hypothesis about the problem. A hypothesis is a *testable idea* for solving the problem. The nice thing about a hypothesis is that it is either true or false, which works well with our Boolean programmer brains: either/or, black or white, true or false.

The key here is to write all of this down. When I was young I never wrote things down because I thought I could keep them all in my head. But that was nonsense; I could not keep them all in my head, and I did not know the ones I had forgotten until my boss at the time asked me a question I could not answer. It is unsettling to realize you have a dumb look on your face in response to a question about something you are working on.

Eventually I developed a system of note taking that allowed me to make this a bit easier. When I have a theory about a problem, I create a note titled THEORY, and write down my idea. Under this, I write up all my tests (which I call TEST, because like any good programmer, I do not want to keep typing HYPOTHESIS). The note-taking system I currently use is Org mode in Emacs, which lets you create sequences that can be tied to hot keys, allowing you to change labels quickly. For bugs, I have labels called BUG, ANALYZED, PATCHED, |, and FIXED, while for hypotheses I have either PROVEN or DISPROVEN.

I always keep both the proven and disproven hypotheses. Why do I keep both? Because that way I know what I tried, and what worked and what failed. This proves to be invaluable when you have a boss with OCD, or, as they like to be called, "detail oriented." By keeping both your successes and failures, you can always go back, say in three months when the code breaks in a disturbingly similar way to the bug you closed, and look at what you tested last time. Maybe one of those hypotheses will prove to be useful, or maybe they will just remind you of the dumb things you tried, so you do not waste time trying them again. Whatever the case, you should store them, backed up, in some version-controlled way. Mine are in my personal source-code repository. You have your own repository, right? Right?!

KV

Related articles on queue.acm.org

Kode Vicious Bugs Out
George Neville-Neil
http://queue.acm.org/detail.cfm?id=1127862

Voyage in the Agile Memeplex
Philippe Kruchten
http://queue.acm.org/detail.cfm?id=1281893

ORM in Dynamic Languages
Chris Richardson
http://queue.acm.org/detail.cfm?id=1394140

George V. Neville-Neil (kv@acm.org) is the proprietor of Neville-Neil Consulting and a member of the *ACM Queue* editorial board. He works on networking and operating systems code for fun and profit, teaches courses on various programming-related subjects, and encourages your comments, quips, and code snips pertaining to his *Communications* column.