Joel on Software

Painless Bug Tracking

by Joel Spolsky

Wanted: Full/Part C++ Product Developer - Programming at LightSpeed Gaming LLC (Cincinnati, OH). See this and other great job listings on the jobs page.

Wednesday, November 08, 2000

TRS-80 Level-I BASIC could only store two string variables, A$ and B$. Similarly, I was born with only two bug-storing-slots in my brain. At any given time, I can only remember two bugs. If you ask me to remember three, one of them will fall on the floor and get swept under the bed with the dust bunnies, who will eat it.

Keeping a database of bugs is one of the hallmarks of a good software team. I never cease to be amazed at how few teams are actually doing this. One of the biggest incorrect facts that programmers consistently seem to believe is that they can remember all their bugs or keep them on post-it notes.

If I can bend your ear a moment, I'd like to explain a pretty painless way to do bug tracking, in the spirit of my previous articles on painless schedules and painless specs.

First of all, you need a real database. On teams of 2 people writing a little bit of code over the course of a long weekend, it's probably OK to use a text file as the database. Anything larger, and you're going to need a real bug tracking database. There are zillions of bug tracking databases you can buy. (Blatant self-promotion: the one we wrote at Fog Creek Software, called FogBUGZ, is web based, pretty easy to use, and quite powerful, if I may say so myself.)

Let's follow a bug around, for the purpose of illustration, from the moment it's born until someone finally puts it out of its misery. We'll follow the famous Bug 1203. Here's what the bug database shows for that bug:

ID	1203
Project	Bee Flogger 2.0
Area	FTP Client
Title	**Uploading file causes FTP server to dump core**
Assigned To	CLOSED
Status	CLOSED (RESOLVED - FIXED)

Priority	2 - Must Fix
Fix For	2.0 Alpha
Version	Build 2019
Computer	Jill's iMac, Mac OS 9.0, 128M RAM, 1024x768 millions of colors
Description	**11/1/2000 Opened by Jill the Very, Very Good Tester**

* Start Bee Flogger

* Create an unnamed document simply containing the letter "a"

* Click on the FTP button on the toolbar

* Try to ftp to your server

BUG: Observe; the ftp server is no longer responding. Indeed ps -augx shows that it is not even running and there is a core dump in /.

EXPECTED: No crash

11/1/2000 Assigned to Willie the Lead Developer by Jill the Very, Very Good Tester

11/2/2000 (Yesterday) RESOLVED - WON'T FIX by Willie the Lead Developer

Not our code, Jill, that's just proftpd which comes with Linux.

11/2/2000 (Yesterday) Reactivated (assigned to Willie the Lead Developer) by Jill the Very, Very Good Tester

That doesn't sound right. I've never been able to crash proftpd when I connect with a normal ftp client. Our code crashes it every single time. Ftp servers don't just "crash".

11/3/2000 (Today) Assigned to Mikey the Programmer by Willie the Lead Developer

Mikey, can you look at this? Maybe your client code is doing something wrong.

11/3/2000 (Today) RESOLVED - FIXED by Mikey the Programmer

I think I was passing the user name instead of the password or something...

11/3/2000 (Today) Reactivated (assigned to Mikey the Programmer) by Jill the Very, Very Good Tester

Still happens in Build 2021.

11/3/2000 (Today) Edited by Mikey the Programmer

Whoa. That's strange. Lemme debug this.

11/3/2000 (Today) Edited by Mikey the Programmer

I'm thinking it might be MikeyStrCpy()...

11/3/2000 (Today) RESOLVED - FIXED by Mikey the Programmer

Ahhh!
FIXED!

11/3/2000 (Today) Closed by Jill the Very, Very Good Tester

Appears fixed in build 2022, so I'll go ahead and close this.

Here's what happened.

Mikey the Programmer is hacking away on the new FTP client feature of his groovy Macintosh software. At some point, because he's feeling frisky, he writes his *own* string-copy function. That'll teach them pesky reusability police! Bwa ha ha!

Bad things happen when you don't reuse code, Mikey. And today, the bad thing that happened was that Mikey forgot to null-terminate the copied string. But he never noticed the problem because most of the time he happened to be copying strings into pre-zeroed memory.

Later that week, Jill the Very, Very Good Tester is banging away at the code, rolling her forehead back and forth on the keyboard or some equally cruel test. (Incidentally, most good testers are named Jill, or some variation like Gillian.) Suddenly something *very* strange happens: the ftp daemon she was testing against *crashed*. Yes, I know it's a Linux machine and Linux machines never crash (no snorting sounds from the slashdot crowd, please) but this dang thing *crashed*. And she wasn't even touching the server, she was just FTPing files to it using Mikey's Mac code.

Now, Jill is a very, very good tester, so she's kept a careful log of what she was doing (the precise pitch and yaw of her head as she rolled it on the keyboard is in her lab book, for example). She reboots everything, starts with a clean machine, repeats the steps, and -- Lo and Behold -- it happens *again!* The Linux ftp daemon crashed *again!* That's *twice in one day, now!* Take that, Linus.

Jill squints at the list of repro steps. There are about 20 steps. Some of them don't seem related. After a bit of experimentation, Jill is able to whittle the problem down to four steps that always cause the same behavior. Now she's ready to file a bug.

Jill enters the new bug in the bug tracking database. Now, just the act of entering a bug requires some discipline: there are good bug reports and bad bug reports.

Three Parts To Every Good Bug Report

And the Lord spake, saying, "First shalt thou take out the Holy Pin. Then, shalt thou count to three, no more, no less. Three shall be the number thou shalt count, and the number of the counting shalt be three. Four shalt thou not count, nor either count thou two, excepting that thou then proceed to three. Five is right out. Once the number three, being the third number, be reached, then lobbest thou thy Holy Hand Grenade of Antioch towards thou foe, who being naughty in my sight, shall snuff it."

-- *Monty Python and the Holy Grail*

It's pretty easy to remember the rule for a good bug report. **Every good bug report needs exactly three things.**

1. Steps to reproduce,
2. What you expected to see, and

3. What you saw instead.

Seems easy, right? Maybe not. As a programmer, people regularly assign me bugs where they left out one piece or another.

If you don't tell me how to repro the bug, I probably will have no idea what you are talking about. "The program crashed and left a smelly turd-like object on the desk." That's nice, honey. I can't do anything about it unless you tell me *what you were doing*. Now, I admit that there are two cases where it's hard to get exact steps to repro. Sometimes you just don't remember, or you're just transcribing a bug from "the field." (By the way, why do they call it "the field"? Is it, like, a field of rye or something? Anyway...) The other time it's OK not to have repro steps is when the bug happens *sometimes* but not *all the time*, but you should still provide repro steps, with a little annotation that says that it doesn't happen too often. In these cases, it's going to be really hard to find the bug, but we can try.

If you don't specify *what you expected to see*, I may not understand why this is a bug. The splash screen has blood on it. So what? I cut my fingers when I was coding it. What did you expect? Ah, you say that the spec required *no blood*! Now I understand why you consider this a bug.

Part three. What you saw instead. If you don't tell me this, I don't know what the bug is. That one is kind of obvious.

Back To Our Story

Anyhoo. Jill enters the bug. In a good bug tracking system it gets automatically assigned to the lead developer for that project. And therein lies the second concept: every bug needs to be assigned to *exactly one person* at all times, until it is closed. A bug is like a hot potato: when it's assigned to you, you are responsible to resolve it, somehow, or assign it to someone *else*.

Willie, the lead developer, looks at the bug, decides it's probably something to do with the ftp server, and resolves it as "won't fix." After all, they didn't *write* the ftp server.

When a bug is resolved, it gets assigned back to the person who opened it. This is a crucial point. It does *not* go away just because a programmer thinks it should. The golden rule is that only the person who opened the bug can close the bug. The programmer can *resolve* the bug, meaning, "hey, I think this is done," but to actually *close* the bug and get it off the books, the original person who opened it needs to confirm that it was actually fixed or agree that it shouldn't be fixed for some reason.

Jill gets an email telling her that the bug is back in her court. She looks at it and reads Willie the Lead Developer's comments. Something doesn't sound right. People have been using this ftp server for years and it doesn't crash. It only crashes when you use Mikey's code. So Jill *reactivates* the bug explaining her position, and the bug goes back to Willie. This time Willie assigns the bug to Mikey to fix.

Mikey studies the bug, thinks long and hard, and completely misdiagnoses the bug. He fixes some altogether different bug, and then resolves the one Jill opened.

The bug is back to Jill, this time marked "RESOLVED-FIXED". Jill tries her repro steps with the latest build, and, lo and behold, the Linux server crashes. She reactivates the bug *again* and assigns it straight back to Mikey.

Mikey is perplexed, but he finally tracks down the source of the bug. (Know what it is yet? I'll leave it as an exercise to the reader. I've given you enough clues!) He fixes it, tests it, and -- Eureka! The repro case no longer crashes the ftp server. Once again, he resolves it as FIXED. Jill also tries the repro steps, discovers that the bug is good 'n' fixed, and closes it out.

Top Ten Tips for Bug Tracking

1. A good tester will always try to reduce the repro steps to the *minimal steps* to reproduce; this is extremely helpful for the programmer who has to find the bug.
2. Remember that the only person who can *close* a bug is the person who opened it in the first place. Anyone can *resolve* it, but only the person who saw the bug can really be sure that what they saw is fixed.
3. There are many ways to resolve a bug. FogBUGZ allows you to resolve a bug as *fixed*, *won't fix*, *postponed*, *not repro*, *duplicate*, or *by design*.
4. *Not Repro* means that nobody could ever reproduce the bug. Programmers often use this when the bug report is missing the repro steps.
5. You'll want to keep careful track of versions. Every build of the software that you give to testers should have a build ID number so that the poor tester doesn't have to retest the bug on a version of the software where it wasn't even supposed to be fixed.
6. If you're a programmer, and you're having trouble getting testers to use the bug database, just *don't accept bug reports by any other method*. If your testers are used to sending you email with bug reports, just bounce the emails back to them with a brief message: "please put this in the bug database. I can't keep track of emails."
7. If you're a tester, and you're having trouble getting programmers to use the bug database, just *don't tell them about bugs* - put them in the database and let the database email them.
8. If you're a programmer, and only some of your colleagues use the bug database, just start assigning them bugs in the database. Eventually they'll get the hint.
9. If you're a manager, and nobody seems to be using the bug database that you installed at great expense, start assigning new features to people using bugs. A bug database is also a great "unimplemented feature" database, too.
10. Avoid the temptation to add new fields to the bug database. Every month or so, somebody will come up with a great idea for a new field to put in the database. You get all kinds of clever ideas, for example, keeping track of the file where the bug was found; keeping track of what % of the time the bug is reproducible; keeping track of how many times the bug occurred; keeping track of which exact versions of which DLLs were installed on the machine where the bug happened. It's very important *not* to give in to these ideas. If you do, your new bug entry screen will end up with a thousand fields that you need to supply, and nobody will want to input bug reports any more. For

> the bug database to work, everybody needs to use it, and if entering bugs "formally" is too much work, people will go *around* the bug database.

If you are developing code, even on a team of one, without an organized database listing all known bugs in the code, you are simply going to ship low quality code. On good software teams, not only is the bug database used universally, but people get into the habit of using the bug database to make their own "to-do" lists, they set their default page in their web browser to the list of bugs assigned to them, and they start wishing that they could assign bugs to the office manager to stock more Mountain Dew.

Next: Big Macs vs. The Naked Chef