



To
Track
Or
Not to
Track

By Lisa Crispin

“Should we track defects so we can learn from them?”

Or does the efficiency of ‘fix ‘em’ outweigh the benefits of recording defects in a tracking system?”



As a tester, I’ve always believed that defect-tracking systems are a necessary tool in the software development toolbox—like source code control and a database. In some situations, I’ve found these systems to be helpful repositories of knowledge; other times they were required to keep defects from falling into a big black hole. I’ve always assumed that every software development team needs some efficient way to track defects, make sure they get fixed, discover whether the same ones are recurring, investigate their underlying causes, and notice if they’re concentrated in certain parts of an application.

But recently, I’ve learned that some software developers, who are applying lessons learned from lean manufacturing, don’t log defects. Basically, a log is an inventory, and the lean community considers any kind of inventory a liability. As Tom and Mary Poppendieck write in their book *Implementing Lean Software Development*:

Defect tracking systems are queues of partially done work, queues of rework if you will. Too often we think that just because a defect is in a queue, it’s OK, we won’t lose track of it. But in the lean paradigm, queues are collection points for waste.

Rather than tracking defects found during development, some software teams simply fix them as soon as they are discovered. They don’t keep any record of the defect except a test written to reproduce the problem, prove that the fix worked, and detect any future regressions. These teams may track bugs found after release but not bugs found “internally” by developers or testers.

But rather than a liability, could a database of defects provide some value? Could software developers

benefit by looking for patterns in the types of defects they create and in so doing improve their practices or processes to prevent them in the future? Should we track defects so we can learn from them? Or does the efficiency of “fix ‘em and forget ‘em” outweigh the benefits of recording defects in a tracking system?

I asked a number of people working “in the trenches” of software development how they approach this question, and I took a hard look at my own team’s experience. Here’s what I discovered. (See the StickyNotes for references).



The Case for Defect Tracking

Although most of our project’s defects are fixed within a day or two, my team records almost all of them in a defect-tracking system. If it’s a bug in newly checked-in code and the programmer fixes it right away, we usually don’t log it. But if it won’t get fixed immediately, we record the steps to reproduce the problem in the tracking system and stick an index card with the bug number and subject on our task board. All bugs found in production also are logged, and if they are a high priority they get stuck on the task board to be fixed that day.

Sometimes the programmers put detailed notes in the defect report about the issue they found and fixed. For example, in a defect report two months ago, a developer recorded his analysis of the problem, noted an immediate solution via a data update, and then explained a more detailed code solution. He also documented other issues we had discussed and our decision that the software was working as designed. It would have been difficult to put that much detail into comments within the code or a test case. This week, a

previously undetected problem surfaced in the same code. Having detailed notes from a related problem saved us a lot of time in analyzing the second issue.

I was surprised that our programmers find the defect-tracking system quite useful. The ability to read clear, detailed steps to reproduce the problem saves them time. If we didn't use defect-tracking software, we could write these details on cards or paper, or perhaps annotate screen prints, but it probably wouldn't be as easy to use. Of course, the programmer who plans to fix the bug could just talk to the tester who found the defect, but what if that tester isn't around at the time the programmer starts working on the bug?

Like many organizations, we have an old and rather ugly legacy system that will continue to haunt us for years to come. The business folk don't want us to spend time fixing the low-priority defects; they'd rather have us work on new features. But if we rewrite part of the old system, it's useful to know its current problems. Besides, every once in a while we have time to repair a few low-priority defects. We couldn't do that without easily retrievable defect records.

Some teams mine their "bugbases" for more information to use root cause analysis to reduce their defect rates. Chris Wheeler recommends this and comments:

There is some value to having project memory in a more durable form than the collective conscience of the team. This is because, in real life, teams do disband, projects do get outsourced to teams in different time zones, products do move into sustaining groups, and sometimes the brains don't go along with the product. Sometimes the tracking system facilitates better collaboration.

Tracking defects doesn't necessarily mean using an automated tool. Dave Rooney worked on a team that took a more visual approach. When testers found a problem, they printed screen shots, wrote down the steps to reproduce the problem, discussed it with a developer, and put their sheets in a pink file folder in the development area. Develop-

ers would triage each defect and work at fixing it into the schedule. Fixed defects were moved to a blue file folder and retested. Dave notes:

The file folders had an interesting visual impact—you could check at a glance how many bugs were being found. If the folder started to get more than a few bug reports, we'd "pull the cord" and stop to figure out as a team what was wrong.

To me, an important quality of any defect-tracking system is low overhead. While many people associate defect tracking with heavyweight, complex systems and processes, it doesn't have to be that way. If your bugbase is weighing you down, look for alternatives. Brad Appleton advises that defect reports should be kept to the bare necessities—and make sure it's really a defect when you report it. He emphasizes that bugbases should enable communication and collaboration, not become "the wall" over which you throw things back and forth. Talking over an issue with another team member might help narrow it down or determine it isn't a bug.

Some industry experts maintain that these days of regulatory compliance à la Sarbanes-Oxley may require us to record historical information such as reported defects and their dispositions. Jim Shore acknowledges that you may need a formal way to track defects. However, he cautions:

I never assume that a database will be necessary until the requirements of the situation prove that it is. Even then, I look for ways to use our existing process to meet the regulatory or organizational requirements. Although some shudder at the informality, archiving red bug cards in a drawer may be enough.

Finally, users may want to know the status of their problem—has it been fixed yet? If not, is it being worked on? Janet Gregory notes:

One of the important uses of a tracking system is for generating client reports. Clients want to know which bugs were fixed and released in the latest version. Most clients want to test and to make

certain the fixes were done to their satisfaction.



Apart from the "lean" idea that an inventory of defects is a liability rather than an asset, proponents of simply fixing bugs as they are found and not recording them at all often find that bugbases may be a dumping ground, cluttered with bugs that never will see the light of day. Our bug database contains hundreds of bugs, mostly from the legacy system, that probably will never get fixed.

Another complaint is that tracking systems don't tend to promote collaboration. As Jim Shore puts it:

Explicitly not providing a database helps create the attitude that bugs are an abnormality. It also removes the temptation to use the database as a substitute for direct, in-person communication.

Ron Jeffries shares this viewpoint: *I have never participated in a really great meeting with people sitting around the issues database.*

The preferred approach to handling defects in the "lean" school is along the lines of what Ron personally does:

1. Write a test showing that the bug exists.
2. Add it to the test suite.
3. Make the test run.
4. Take no other action.

Now the defect not only is fixed, but if that particular piece of code fails again the same way, we have a regression test to catch it. Yes, other scenarios still may produce unforeseen bugs, but preventing regressions will give you more time to look for those.

If you think the whole idea of "fix and forget" is loony, at least in your own situation, take a fresh look at why you're tracking defects. When I fretted that without recording a defect in a tracking system I'd forget the details, Michael Bolton offered me this advice:

“...the important debate isn't over whether we ought to track defects that already have been discovered. It's about how we can learn to minimize defects in the code we deliver in the future.”



If you have written a defect down or recorded it in some way, you have an issue tracker. The questions then become “What do you value in an issue tracker?” and “Whose needs are you trying to satisfy?” If you're working on a project where the team is dispersed worldwide, a Moleskine notebook is probably not the place to record defects for the entire team; JIRA might be a better choice. If you're taking notes about a bug, preparing for a face-to-face conversation with a developer ten minutes hence, JIRA might be overkill and the Moleskine might be just the thing. If you're going to pass your notes to the developer, an index card or two might be preferable, since you don't want to give her your whole Moleskine. If you want to keep your notes in order, the Moleskine is probably better than index cards, because being unbound they might get out of order. If you want to record test ideas as they occur to you, free form, JIRA doesn't seem like a good choice; if you want to be able to filter out all the defects assigned to a particular developer, JIRA can produce a report in an instant. What do you want to do today? You might want to use multiple tools to accomplish multiple tasks.

(See the StickyNotes for more on Moleskines and JIRA.)



To Track or Not to Track?

That is the Question

Even teams that “fix and forget” are likely to track production bugs. Most businesses want reports with information about defects. Andreas Zeller's book *Why Programs Fail* asserts that as a manager you must be able to answer questions such as:

- Which problems currently are open?
- Which are the most severe problems?
- Did similar problems occur in the past?

So while you still might track production bugs, your focus should be on bug prevention. A large number of bugs—whether you're tracking them or not—is a red flag that needs investigation. Mary Poppendieck shared these observations with me when I asked her about my team's issues with giving up the defect tracking system:

The objective is not to get rid of a defect tracking system—it is to not need a defect tracking system. There is a big difference. The trick is to expose a defect the moment it occurs. Now this is far easier said than done, but it should be the team's objective.

In other words, stop worrying about the defect database and start

worrying about why you are still creating code where defects are discovered a significant amount of time after the code has been written.

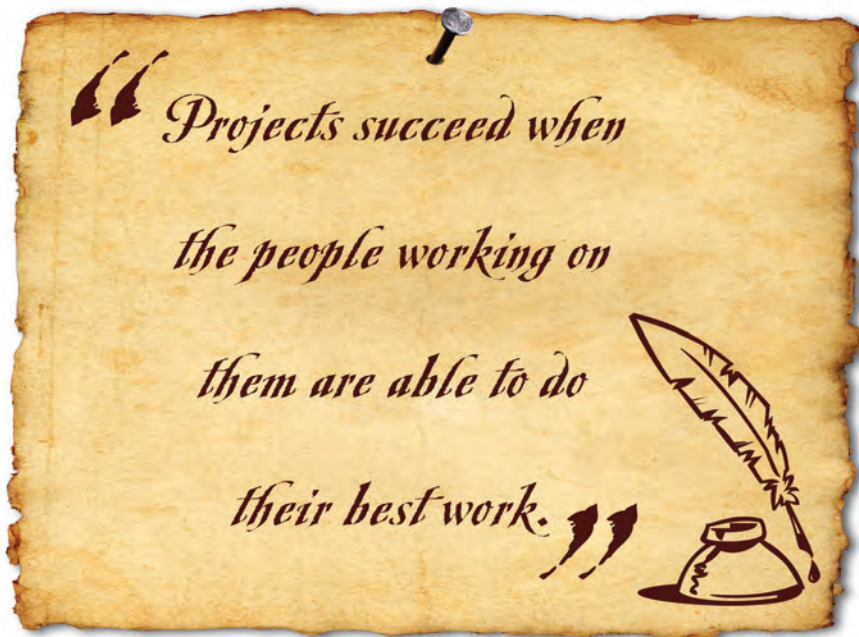
Jean McAuliffe echoes these sentiments:

The goal is defect elimination, not defect discovery.

That's a lofty goal, but you can realistically strive to reduce the number of defects found during testing or in production. As Mary and Jean explained, eliminating many defects requires disciplined practices, such as test-driven development (writing tests in concert with the code), continuous integration, and designing both architecture and code for testability. The time span between coding, integrating, and testing must be short enough that bugs are caught while their cause is still obvious, making them easier to fix. Techniques such as automating regression testing and taking time for diligent exploratory testing as soon as the code is written help teams lower their defect rate.

Jean pointed out that, like me, a lot of testers “grew up” when defect-tracking systems were often the only way to communicate issues to the programmers. I expect this is still true for many testers. But for teams that have adopted disciplined practices and have built solid processes there might be simpler ways to deal with defects.

If we don't track all of our defects, what happens to the knowledge of what



defects occurred and where and how they were fixed? As Ron Jeffries pointed out to me, other ways exist for a team to learn from mistakes and improve its practices and processes. Retrospectives are an excellent tool for reviewing what's been working well and what hasn't, identifying problem areas such as a high number of defects in one area of the application, and identifying actions to address them. If you do this on a regular basis, you might not have much to learn from a defect database. There is simply no substitute for face-to-face communication.

Exploring this topic led me to understand that the important debate isn't over whether we ought to track defects that already have been discovered. It's about how we can learn to minimize defects in the code we deliver in the future. If you're working on a buggy legacy system, root cause analysis of defects in a bugbase could help your team identify high risk areas and focus your efforts on rewriting them in a testable, high-quality fashion. If you're on a "greenfield" project with a collocated project team and write code in a way that produces few defects, why take on the overhead of a defect-tracking system?

Even if you have a defect database, you might sometimes choose not to use it. Janet Gregory offers these words of wisdom:

A defect-tracking system is not meant as a communication tool. Nothing can take the place of discussions with the developers. The best thing a tester can do is to talk

to the developer as soon as a problem is found. If they can fix it immediately, there is no need to enter a report, because you have a test that will catch it if it happens again. The issue is dealt with quickly and avoids waste.

If you use a defect-tracking system, make sure it doesn't replace direct communication. If you're spending a lot of time trying to find out if someone has fixed a defect yet, you may need a defect-tracking system (or a different one, if you have one and still can't determine the status of defects). If you already have a defect database, consider mining it for information about where defects cluster, and focus on cleaning up the related code. Most importantly, make sure all your tools—bug trackers included—fit your needs and help your team improve, rather than getting in your way.

Whether or not you maintain a defect database, see if your team can fix defects as soon as they are discovered, and make sure every defect has a test written for it that will catch future regressions as well as provide information about the defect. Fixing bugs as soon as they arise is generally a good practice for everyone. Surprisingly (at least, it was a surprise to me), your business managers may not want you to fix every single bug found. They may prefer that the development team concentrate on delivering new, valuable features, instead of spending time on hard-to-reproduce bugs or problems that users can work around.

Most importantly, your team should

continually review its progress, identify problem areas, and collaborate on ways to improve. If you already have a big backlog of bugs, they might provide some useful information to help you focus your efforts. There are many practices that may help prevent bugs. Write testable code, rewrite buggy areas test-first. Try shortening the cycle of coding, integrating, and testing so that programmers get quick feedback about code quality. Involve business experts so that requirements are well understood.

In the case of our team, we may find in a year or two that we've gotten really good at writing defect-free code, we've rewritten the really icky parts of our legacy system, and we have so few production defects that we don't feel the need to log them in an online database. We can decide then to stop using it, or we can decide that it's not a lot of extra work to log defects in the system and that it's a useful knowledgebase to keep around.

We need to keep our focus on the right target. We want to deliver the best quality code that we can, and we want to deliver value to the business in a timely manner. Projects succeed when the people working on them are able to do their best work. Our focus should be on improving communication and facilitating collaboration. If we encounter many defects, we need to investigate the real source of the problem. If we need a defect-tracking database to do that, so be it. If our team works more efficiently by documenting defects in test cases and fixing them right away, let's do that. If some combination of the two approaches supports our ability to improve continually, then that is right for us. **{end}**

Since 2000, Lisa Crispin has been a tester on agile teams developing Web-based applications. She co-authored Testing Extreme Programming (Addison-Wesley, 2002) with Tip House and is a regular contributor to Better Software magazine. Read more about Lisa's work at lisa.crispin.home.att.net.

Sticky Notes

For more on the following topics go to www.StickyMinds.com/bettersoftware.

- References
- Moleskines
- JIRA