Joel on Software

The Joel Test: 12 Steps to Better Code

by Joel Spolsky

Wednesday, August 09, 2000

Have you ever heard of SEMA? It's a fairly esoteric system for measuring how good a software team is. No, *wait! Don't follow that link!* It will take you about six years just to *understand* that stuff. So I've come up with my own, highly irresponsible, sloppy test to rate the quality of a software team. The great part about it is that it takes about 3 minutes. With all the time you save, you can go to medical school.

The Joel Test

1. Do you use source control?
2. Can you make a build in one step?
3. Do you make daily builds?
4. Do you have a bug database?
5. Do you fix bugs before writing new code?
6. Do you have an up-to-date schedule?
7. Do you have a spec?
8. Do programmers have quiet working conditions?
9. Do you use the best tools money can buy?
10. Do you have testers?
11. Do new candidates write code during their interview?
12. Do you do hallway usability testing?

The neat thing about The Joel Test is that it's easy to get a quick **yes** or **no** to each question. You don't have to figure out lines-of-code-per-day or average-bugs-per-inflection-point. Give your team 1 point for each "yes" answer. The bummer about The Joel Test is that you *really shouldn't* use it to make sure that your nuclear power plant software is safe.

A score of 12 is perfect, 11 is tolerable, but 10 or lower and you've got serious problems. The truth is that most software organizations are running with a score of 2 or 3, and they need *serious* help, because companies like Microsoft run at 12 full-time.

Of course, these are not the only factors that determine success or failure: in particular, if you have a great software team working on a product that nobody wants, well, people aren't going to want it. And it's possible to imagine a team of "gunslingers" that doesn't do any of this stuff that still manages to produce incredible software that changes the world. But, all else being equal, if you get these 12 things right, you'll have a disciplined team that can consistently deliver.

1. Do you use source control?

I've used commercial source control packages, and I've used CVS, which is free, and let me tell you, CVS is *fine*. But if you don't have source control, you're going to stress out trying to get programmers to work together. Programmers have no way to know what other people did. Mistakes can't be rolled back easily. The other neat thing about source control systems is that the source code itself is checked out on every programmer's hard drive -- I've never heard of a project using source control that lost a lot of code.

2. Can you make a build in one step?

By this I mean: how many steps does it take to make a shipping build from the latest source snapshot? On good teams, there's a single script you can run that does a full checkout from

scratch, rebuilds every line of code, makes the EXEs, in all their various versions, languages, and #ifdef combinations, creates the installation package, and creates the final media -- CDROM layout, download website, whatever.

If the process takes any more than one step, it is prone to errors. And when you get closer to shipping, you want to have a very fast cycle of fixing the "last" bug, making the final EXEs, etc. If it takes 20 steps to compile the code, run the installation builder, etc., you're going to go crazy and you're going to make silly mistakes.

For this very reason, the last company I worked at switched from WISE to InstallShield: we *required* that the installation process be able to run, from a script, automatically, overnight, using the NT scheduler, and WISE couldn't run from the scheduler overnight, so we threw it out. (The kind folks at WISE assure me that their latest version does support nightly builds.)

3. Do you make daily builds?

When you're using source control, sometimes one programmer accidentally checks in something that breaks the build. For example, they've added a new source file, and everything compiles fine on their machine, but they forgot to add the source file to the code repository. So they lock their machine and go home, oblivious and happy. But nobody else can work, so they have to go home too, unhappy.

Breaking the build is so bad (and so common) that it helps to make daily builds, to insure that no breakage goes unnoticed. On large teams, one good way to insure that breakages are fixed right away is to do the daily build every afternoon at, say, lunchtime. Everyone does as many checkins as possible before lunch. When they come back, the build is done. If it worked, great! Everybody checks out the latest version of the source and goes on working. If the build failed, you fix it, but everybody can keep on working with the pre-build, unbroken version of the source.

On the Excel team we had a rule that whoever broke the build, as their "punishment", was responsible for babysitting the builds until someone else broke it. This was a good incentive not to break the build, and a good way to rotate everyone through the build process so that everyone learned how it worked.

Read more about daily builds in my article Daily Builds are Your Friend.

4. Do you have a bug database?

I don't care what you say. If you are developing code, even on a team of one, without an organized database listing all known bugs in the code, you are going to ship low quality code. Lots of programmers think they can hold the bug list in their heads. Nonsense. I can't remember more than two or three bugs at a time, and the next morning, or in the rush of shipping, they are forgotten. You absolutely have to keep track of bugs formally.

Bug databases can be complicated or simple. A minimal useful bug database must include the following data for every bug:

- complete steps to reproduce the bug
- expected behavior
- observed (buggy) behavior
- who it's assigned to
- whether it has been fixed or not

If the complexity of bug tracking software is the only thing stopping you from tracking your bugs, just make a simple 5

column table with these crucial fields and *start using it*.

For more on bug tracking, read Painless Bug Tracking.

5. Do you fix bugs before writing new code?
The very first version of Microsoft Word for Windows was considered a "death march" project. It took forever. It kept slipping. The whole team was working ridiculous hours, the project was delayed again, and again, and again, and the stress was incredible. When the dang thing finally shipped, years late, Microsoft sent the whole team off to Cancun for a vacation, then sat down for some serious soul-searching.

What they realized was that the project managers had been so insistent on keeping to the "schedule" that programmers simply rushed through the coding process, writing extremely bad code, because the bug fixing phase was not a part of the formal schedule. There was no attempt to keep the bug-count down. Quite the opposite. The story goes that one programmer, who had to write the code to calculate the height of a line of text, simply wrote "return 12;" and waited for the bug report to come in about how his function is not always correct. The schedule was merely a checklist of features waiting to be turned into bugs. In the post-mortem, this was referred to as "infinite defects methodology".

To correct the problem, Microsoft universally adopted something called a "zero defects methodology". Many of the programmers in the company giggled, since it sounded like management thought they could reduce the bug count by executive fiat. Actually, "zero defects" meant that at any given time, the highest priority is to eliminate bugs *before* writing any new code. Here's why.

In general, the longer you wait before fixing a bug, the costlier (in time and money) it is to fix.

For example, when you make a typo or syntax error that the compiler catches, fixing it is basically trivial.

When you have a bug in your code that you see the first time you try to run it, you will be able to fix it in no time at all, because all the code is still fresh in your mind.

If you find a bug in some code that you wrote a few days ago, it will take you a while to hunt it down, but when you reread the code you wrote, you'll remember everything and you'll be able to fix the bug in a reasonable amount of time.

But if you find a bug in code that you wrote a few *months* ago, you'll probably have forgotten a lot of things about that code, and it's much harder to fix. By that time you may be fixing somebody *else's* code, and they may be in Aruba on vacation, in which case, fixing the bug is like science: you have to be slow, methodical, and meticulous, and you can't be sure how long it will take to discover the cure.

And if you find a bug in code that has *already shipped*, you're going to incur incredible expense getting it fixed.

That's one reason to fix bugs right away: because it takes less time. There's another reason, which relates to the fact that it's easier to *predict* how long it will take to write new code than to fix an existing bug. For example, if I asked you to predict how long it would take to write the code to sort a list, you could give me a pretty good estimate. But if I asked you how to predict how long it would take to fix that bug where your code doesn't work if Internet Explorer 5.5 is installed, you can't even *guess*,

because you don't know (by definition) what's *causing* the bug. It could take 3 days to track it down, or it could take 2 minutes.

What this means is that if you have a schedule with a lot of bugs remaining to be fixed, the schedule is unreliable. But if you've fixed all the *known* bugs, and all that's left is new code, then your schedule will be stunningly more accurate.

Another great thing about keeping the bug count at zero is that you can respond much faster to competition. Some programmers think of this as keeping the product *ready to ship* at all times. Then if your competitor introduces a killer new feature that is stealing your customers, you can implement just that feature and ship on the spot, without having to fix a large number of accumulated bugs.

6. Do you have an up-to-date schedule?
Which brings us to schedules. If your code is at all important to the business, there are lots of reasons why it's important to the business to know when the code is going to be done. Programmers are notoriously crabby about making schedules. "It will be done when it's done!" they scream at the business people.

Unfortunately, that just doesn't cut it. There are too many planning decisions that the business needs to make well in advance of shipping the code: demos, trade shows, advertising, etc. And the only way to do this is to have a schedule, and to keep it up to date.

The other crucial thing about having a schedule is that it forces you to decide what features you are going to do, and then it forces you to pick the least important features and *cut them* rather than slipping into featuritis (a.k.a. scope creep).

Keeping schedules does not have to be hard. Read my article Painless Software Schedules, which describes a simple way to make great schedules.

7. Do you have a spec?
Writing specs is like flossing: everybody agrees that it's a good thing, but nobody does it.

I'm not sure why this is, but it's probably because most programmers hate writing documents. As a result, when teams consisting solely of programmers attack a problem, they prefer to express their solution in code, rather than in documents. They would much rather dive in and write code than produce a spec first.

At the design stage, when you discover problems, you can fix them easily by editing a few lines of text. Once the code is written, the cost of fixing problems is dramatically higher, both emotionally (people hate to throw away code) and in terms of time, so there's resistance to actually fixing the problems. Software that wasn't built from a spec usually winds up badly designed and the schedule gets out of control. This seems to have been the problem at Netscape, where the first four versions grew into such a mess that management stupidly decided to throw out the code and start over. And then they made this mistake all over again with Mozilla, creating a monster that spun out of control and took *several years* to get to alpha stage.

My pet theory is that this problem can be fixed by teaching programmers to be less reluctant writers by sending them off to take an intensive course in writing. Another solution is to hire smart program managers who produce the written spec. In

either case, you should enforce the simple rule "no code without spec".

Learn all about writing specs by reading my 4-part series.

8. Do programmers have quiet working conditions?
There are extensively documented productivity gains provided by giving knowledge workers space, quiet, and privacy. The classic software management book Peopleware documents these productivity benefits extensively.

Here's the trouble. We all know that knowledge workers work best by getting into "flow", also known as being "in the zone", where they are fully concentrated on their work and fully tuned out of their environment. They lose track of time and produce great stuff through absolute concentration. This is when they get all of their productive work done. Writers, programmers, scientists, and even basketball players will tell you about being in the zone.

The trouble is, getting into "the zone" is not easy. When you try to measure it, it looks like it takes an average of 15 minutes to start working at maximum productivity. Sometimes, if you're tired or have already done a lot of creative work that day, you just can't get into the zone and you spend the rest of your work day fiddling around, reading the web, playing Tetris.

The other trouble is that it's so easy to get knocked *out* of the zone. Noise, phone calls, going out for lunch, having to drive 5 minutes to Starbucks for coffee, and interruptions by coworkers -- *especially* interruptions by coworkers -- all knock you out of the zone. If a coworker asks you a question, causing a 1 minute interruption, but this knocks you out of the zone badly enough that it takes you half an hour to get productive again, your overall productivity is in serious trouble. If you're in a noisy bullpen environment like the type that caffeinated dotcoms love to create, with marketing guys screaming on the phone next to programmers, your productivity will plunge as knowledge workers get interrupted time after time and never get into the zone.

With programmers, it's especially hard. Productivity depends on being able to juggle a lot of little details in short term memory all at once. Any kind of interruption can cause these details to come crashing down. When you resume work, you can't remember any of the details (like local variable names you were using, or where you were up to in implementing that search algorithm) and you have to keep looking these things up, which slows you down a lot until you get back up to speed.

Here's the simple algebra. Let's say (as the evidence seems to suggest) that if we interrupt a programmer, even for a minute, we're really blowing away 15 minutes of productivity. For this example, lets put two programmers, Jeff and Mutt, in open cubicles next to each other in a standard Dilbert veal-fattening farm. Mutt can't remember the name of the Unicode version of the strcpy function. He could look it up, which takes 30 seconds, or he could ask Jeff, which takes 15 seconds. Since he's sitting right next to Jeff, he asks Jeff. Jeff gets distracted and loses 15 minutes of productivity (to save Mutt 15 seconds).

Now let's move them into separate offices with walls and doors. Now when Mutt can't remember the name of that function, he could look it up, which still takes 30 seconds, or he could ask Jeff, which now takes 45 seconds and involves standing up (not an easy task given the average physical fitness of programmers!). So he looks it up. So now Mutt loses 30

seconds of productivity, but we save 15 minutes for Jeff. Ahhh!

9. Do you use the best tools money can buy?

Writing code in a compiled language is one of the last things that still can't be done instantly on a garden variety home computer. If your compilation process takes more than a few seconds, getting the latest and greatest computer is going to save you time. If compiling takes even 15 seconds, programmers will get bored while the compiler runs and switch over to reading The Onion, which will suck them in and kill hours of productivity.

Debugging GUI code with a single monitor system is painful if not impossible. If you're writing GUI code, two monitors will make things much easier.

Most programmers eventually have to manipulate bitmaps for icons or toolbars, and most programmers don't have a good bitmap editor available. Trying to use Microsoft Paint to manipulate bitmaps is a joke, but that's what most programmers have to do.

At my last job, the system administrator kept sending me automated spam complaining that I was using more than ... get this ... 220 megabytes of hard drive space on the server. I pointed out that given the price of hard drives these days, the cost of this space was significantly less than the cost of the *toilet paper* I used. Spending even 10 minutes cleaning up my directory would be a fabulous waste of productivity.

Top notch development teams don't torture their programmers. Even minor frustrations caused by using underpowered tools add up, making programmers grumpy and unhappy. And a grumpy programmer is an unproductive programmer.

To add to all this... programmers are easily bribed by giving them the coolest, latest stuff. This is a far cheaper way to get them to work for you than actually paying competitive salaries!

10. Do you have testers?

If your team doesn't have dedicated testers, at least one for every two or three programmers, you are either shipping buggy products, or you're wasting money by having $100/hour programmers do work that can be done by $30/hour testers. Skimping on testers is such an outrageous false economy that I'm simply blown away that more people don't recognize it.

Read Top Five (Wrong) Reasons You Don't Have Testers, an article I wrote about this subject.

11. Do new candidates write code during their interview?

Would you hire a magician without asking them to show you some magic tricks? Of course not.

Would you hire a caterer for your wedding without tasting their food? I doubt it. (Unless it's Aunt Marge, and she would hate you for*ever* if you didn't let her make her "famous" chopped liver cake).

Yet, every day, programmers are hired on the basis of an impressive resumé or because the interviewer enjoyed chatting with them. Or they are asked trivia questions ("what's the difference between CreateDialog() and DialogBox()?") which could be answered by looking at the documentation. You don't care if they have memorized thousands of trivia about programming, you care if they are able to produce code. Or,

even worse, they are asked "AHA!" questions: the kind of questions that seem easy when you know the answer, but if you don't know the answer, they are impossible.

Please, just *stop doing this*. Do whatever you want during interviews, but make the candidate *write some code*. (For more advice, read my Guerrilla Guide to Interviewing.)

12. Do you do hallway usability testing?

A *hallway usability test* is where you grab the next person that passes by in the hallway and force them to try to use the code you just wrote. If you do this to five people, you will learn 95% of what there is to learn about usability problems in your code.

Good user interface design is not as hard as you would think, and it's crucial if you want customers to love and buy your product. You can read my free online book on UI design, a short primer for programmers.

But the most important thing about user interfaces is that if you show your program to a handful of people, (in fact, five or six is enough) you will quickly discover the biggest problems people are having. Read Jakob Nielsen's article explaining why. Even if your UI design skills are lacking, as long as you force yourself to do hallway usability tests, which cost nothing, your UI will be much, much better.

Next: Wasting Money on Cats

About the author. I'm Joel Spolsky, co-founder of Fog Creek Software, a New York company that proves that you can treat programmers well and still be highly profitable. Programmers get private offices, free lunch, and work 40 hours a week. Customers only pay for software if they're delighted. We make Trello, insanely simple project management, FogBugz, an enlightened bug tracker designed to help great teams develop brilliant software, and Kiln, which simplifies source control. I'm also the co-founder and CEO of Stack Exchange. More about me.

joel@joelonsoftware.com