Joel on Software

Painless Functional Specifications - Part 1: Why Bother?

by Joel Spolsky

Monday, October 02, 2000

When The Joel Test first appeared, one of the biggest sore points readers reported had to do with writing specs. It seems that specs are like flossing: everybody knows they should be writing them, but nobody does.

Why won't people write specs? People claim that it's because they're saving time by skipping the spec-writing phase. They act as if spec-writing was a luxury reserved for NASA space shuttle engineers, or people who work for giant, established insurance companies. Balderdash. First of all, failing to write a spec is the *single biggest unnecessary risk* you take in a software project. It's as stupid as setting off to cross the Mojave desert with just the clothes on your back, hoping to "wing it." Programmers and software engineers who dive into code without writing a spec tend to think they're cool gunslingers, shooting from the hip. They're not. They are terribly unproductive. They write bad code and produce shoddy software, and they threaten their projects by taking giant risks which are completely uncalled for.

I believe that on any non-trivial project (more than about 1 week of coding or more than 1 programmer), if you don't have a spec, you will *always* spend more time and create lower quality code. Here's why.

The most important function of a spec is to *design the program*. Even if you are working on code all by yourself, and you write a spec solely for your own benefit, the act of writing the spec -- describing how the program works in minute detail -- will force you to actually *design* the program.

Let's visit two imaginary programmers at two companies. Speedy, at Hasty Bananas Software, never writes specs. "Specs? We don't need no stinkin' specs!" At the same time, Mr. Rogers, over at The

Well-Tempered Software Company, refuses to write code until the spec is completely nailed down. These are only two of my many imaginary friends.

Speedy and Mr. Rogers have one thing in common: they are both in charge of backwards compatibility for version 2.0 of their respective products.

Speedy decides that the best way to provide backwards compatibility is to write a converter which simply converts 1.0 version files into 2.0 version files. She starts banging that out. Type, type, type. Clickety clickety clack. Hard drives spin. Dust flies. After about 2 weeks, she has a reasonable converter. But Speedy's customers are unhappy. Speedy's code will force them to upgrade everyone in the company at once to the new version. Speedy's biggest customer, Nanner Splits Unlimited, refuses to buy the new software. Nanner Splits needs to know that version 2.0 will still be able to work on version 1.0 files *without* converting them. Speedy decides to write a *backwards* converter and then hook it into the "save" function. It's a bit of a mess, because when you use a version 2.0 feature, it *seems* to work, until you go to save the file in 1.0 format. Only then are you told that the feature you used half an hour ago doesn't work in the old file format. So the backwards converter took another two weeks to write, and it don't work so nice. Elapsed time, 4 weeks.

Now, Mr. Rogers over at Well-Tempered Software Company (colloquially, "WellTemperSoft") is one of those nerdy organized types who *refuses* to write code until he's got a spec. He spends about 20 minutes designing the backwards compatibility feature the same way Speedy did, and comes up with a spec that basically says:

- When opening a file created with an older version of the product, the file is converted to the new format.

The spec is shown to the customer, who says "wait a minute! We don't want to switch everyone at once!" So Mr. Rogers thinks some more, and amends the spec to say:

- When opening a file created with an older version of the product, the file is converted to the new format in memory. When saving this file, the user is given the option to convert it back.

Another 20 minutes have elapsed.

Mr. Rogers' boss, an object nut, looks at this and thinks something might be amiss. He suggests a different architecture.

- The code will be factored to use two interfaces: V1 and V2. V1 contains all the version one features, and V2, which inherits from V1, adds all the new features. Now V1::Save can handle the backwards compatibility while V2::Save can be used to save all the new stuff. If you've opened a V1 file and try to use V2 functionality, the program can warn you right away, and you will have to either convert the file or give up the new functionality.

20 more minutes.

Mr. Rogers is grumpy. This refactoring will take 3 weeks, instead of the 2 weeks he originally estimated! But it does solve *all* the customer problems, in an elegant way, so he goes off and does it.

Total elapsed time for Mr. Rogers: 3 weeks and 1 hour. Elapsed time for Speedy: 4 weeks, but Speedy's code is not as good.

The moral of the story is that with a contrived example, you can prove anything. Oops. No, that's not what I meant to say. The moral of the story is that when you design your product in a human language, it only takes a few minutes to try thinking about several possibilities, revising, and improving your design. Nobody feels bad when they delete a paragraph in a word processor. But when you design your product in a programming language, it takes *weeks* to do iterative designs. What's worse, a programmer who's just spend 2 weeks writing some code is going to be quite attached to that code, no matter how wrong it is. Nothing Speedy's boss or customers could say would convince her to throw away her beautiful converting code, even though that didn't represent the best architecture. As a result, the final product tends to be a compromise between the initial, wrong design and the ideal design. It was "the best design we could get, given that we'd already written all this code and we just didn't want to throw it away." Not quite as good as "the best design we could get, period."

So that's giant reason number one to write a spec. Giant reason number two is to *save time communicating*. When you write a spec, you only have to communicate how the program is supposed to work *once*. Everybody on the team can just read the spec. The QA people read it so that they know how the program is supposed to work and they know what to test for. The marketing people use it to write their vague vaporware white papers to throw up on the web site about products that haven't been created yet. The business development people misread it to spin weird fantasies about how the product will cure baldness and warts and stuff, but it gets investors, so that's OK. The developers read it so that they know what code to write. The customers read it to make sure the developers are building a product that they would want to pay for. The technical writers read it and write a nice manual (that gets lost or thrown away, but that's a different story). The managers read it so that they can look like they know what's going on in management meetings. And so on.

When you don't have a spec, all this communication still happens, *because it has to*, but it happens *ad hoc*. The QA people fool around with the program willy-nilly, and when something looks odd, they go and interrupt the programmers *yet again* to ask them *another* stupid question about how the thing is supposed to work. Besides the fact that this ruins the programmers' productivity, the programmers tend to give the answer that corresponds to what they wrote in the code, rather than the "right answer." So the QA people are really testing the program against the program rather than the program against the design, which would be, um, a *little bit* more useful.

When you don't have a spec, what happens with the poor technical writers is the funniest (in a sad kind of way). Tech writers often don't have the political clout to interrupt programmers. In many companies, if tech writers get in the habit of interrupting programmers to ask how something is supposed to work, the programmers go to their managers and cry about how they can't get any work done because of these [expletive deleted] *writers*, and could they *please* keep them *away*, and the managers, trying to improve productivity, forbid the tech writers to waste *any more* of their *precious* programmers' time. You can always tell these companies, because the help files and the manuals don't give you any more information than you can figure out from the screen. When you see a message on a screen which says

- Would you like to enable LRF-1914 support?

... and you click "Help", a tragicomic help topic comes up which says something like

- Allows you to choose between LRF-1914 support (default) or no LRF-1914 support. If you want LRF-1914 support, choose "Yes" or press "Y". If you don't want LRF-1914 support, choose "No" or press "N".

Um, thanks. It's pretty obvious here that the technical writer was trying to cover up the fact that they *didn't know what LRF-1914 support is*. They couldn't ask the programmer, because (a) they're embarrassed, or (b) the programmer is in Hyderabad and they're in London, or (c) they have been prohibited by management from interrupting the programmer, or any other number of corporate pathologies too numerous to mention, but the fundamental problem is that there *wasn't a spec*.

Number *three* giant important reason to have a spec is that without a detailed spec, it's impossible to make a schedule. Not having a schedule is OK if it's your PhD and you plan to spend 14 years on the thing, or if you're a programmer working on the next Duke Nukem and *we'll ship when we're good and ready*. But for almost any kind of real business, you just have to know how long things are going to take, because developing a product costs *money*. You wouldn't buy a pair of *jeans* without knowing what the price is, so how can a responsible business decide whether to build a product without knowing how long it will take and, therefore, how much it will cost? For more on scheduling, read Painless Software Schedules.

A terribly common error is having a debate over how something should be designed, and then *never resolving the debate*. Brian Valentine, the lead developer on Windows 2000, was famous for his motto "Decisions in 10 minutes or less, or the next one is free."

In too many programming organizations, every time there's a design debate, nobody ever manages to make a *decision*, usually for political reasons. So the programmers only work on uncontroversial stuff. As time goes on, all the hard decisions are pushed to the end. *These are the most likely projects to fail.* If you are starting a new company around a new technology and you notice that your company is constitutionally incapable of making decisions, you might as well close down now and return the money to the investors, because you ain't never gonna ship nothing.

Writing a spec is a great way to nail down all those irritating design decisions, large and small, that get covered up if you don't have a spec. Even small decisions can get nailed down with a spec. For example, if you're building a web site with membership, you might all agree that if the user forgets their password, you'll mail it to them. Great. But that's not enough to write the code. To write the code, you need to know the actual *words* in that email. At most companies, programmers aren't trusted with words that a user might actually see (and for good reason, much of the time). So a marketing person or a PR person or some other English major is likely to be required to come up with the precise wording of the message. "Dear Shlub, Here's the password you forgot. Try not to be so careless in the future." When you force yourself to write a *good, complete* spec (and I'll talk a lot more about that soon), you notice all these things and you either fix them or at least you mark them with a big red flag.

OK. We're on the same page now. Specs are motherhood and apple pie. I suspect that most people understand this, and my rants, while amusing, aren't teaching you anything new. So why *don't* people write specs? It's not to save time, because it *doesn't*, and I think most coders recognize this. (In most organizations, the only "specs" that exist are staccato, one page text documents that a programmer banged out in Notepad *after* writing the code and *after* explaining that damn feature to the three hundredth person.)

I think it's because so many people don't like to write. Staring at a blank screen is horribly frustrating. Personally, I overcame my fear of writing by taking a class in college that required a 3-5 page essay once a week. Writing is a muscle. The more you write, the more you'll be able to write. If you need to write specs and you can't, start a journal, create a weblog, take a creative writing class, or just write a nice letter to every relative and college roommate you've blown off for the last 4 years. Anything that involves putting words down on paper will improve your spec writing skills. If you're a software development manager and the people who are supposed to be writing specs aren't, send them off for one of those two week creative writing classes in the mountains.

If you've never worked in a company that does functional specifications, you may never have seen one. In the next part of this series, I'll show you a short, sample spec for you to check out, and we'll talk about what a good spec needs to have. Read on!

Next: Painless Functional Specifications - Part 2: What's a Spec?

About the author. I'm Joel Spolsky, co-founder of Trello and Fog Creek Software, and CEO of Stack Exchange. More about me.