Joel on Software

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

by Joel Spolsky

Tuesday, October 03, 2000

(Have you already read part one? If not, that's here .)

This series of articles is about *functional specifications*, not *technical* specifications. People get these mixed up. I don't know if there's any standard terminology, but here's what *I* mean when I use these terms.

1. A *functional specification* describes how a product will work entirely from the user's perspective. It doesn't care how the thing is implemented. It talks about features. It specifies screens, menus, dialogs, and so on.
2. A *technical specification* describes the internal implementation of the program. It talks about data structures, relational database models, choice of programming languages and tools, algorithms, etc.

When you design a product, inside and out, the most important thing is to nail down the user experience. What are the screens, how do they work, what do they do. Later, you worry about how to get from here to there. There's no use arguing about what programming language to use before you've decided what your product is going to *do*. In this series, I'm only talking about *functional specifications*.

I've written a short sample spec which should give you an idea for what a good functional specification looks like. Before we go further, please read the sample spec.

Did you read it?

No you didn't. Go read it now and then come back, so we can talk more about what a good spec should and shouldn't have in it. I'll wait here for you. Thanks.

(waiting patiently...)

Ah, good. You're back.

Here are some of the things I put in every spec.

A disclaimer. Pure self defense. If you put a paragraph saying something like "This spec is not complete", people won't come into your office to bite your head off. As time goes on, when the spec starts to be complete, you can change it to say "this spec is complete, to the best of my knowledge, but if I forgot something, please tell me." Which reminds me, every spec needs:

An author. One author. Some companies think that the spec should be written by a *team*. If you've ever tried group writing, you know that there is no worse torture. Leave the group writing to the management consulting firms with armies of newly minted Harvard-educated graduates who need to do a ton of busywork so that they can justify their huge fees. Your specs should be owned and written by *one person*. If you have a big product, split it up into areas and give each area to a different person to spec separately. Other companies think that it's egotistic or not "good teamwork" for a person to "take credit" for a spec by putting their name on it. *Nonsense*. People should take *responsibility* and *ownership* of the things that they specify. If something's wrong with the spec, there should be a designated spec owner, with their name printed right there on the spec, who is responsible for fixing it.

Scenarios. When you're designing a product, you need to have some real live scenarios in mind for how people are going to use it. Otherwise you end up designing a product that doesn't correspond to any real-world usage (like the Cue?Cat). Pick your product's audiences and imagine a fictitious, totally imaginary but totally *stereotypical* user from each audience who uses the product in a totally typical way. Chapter 9 of my UI design book (available online for free) talks about creating fictional users and scenarios. This is where you put them. The more vivid and realistic the scenario, the better a job you will do designing a product for your real or imagined users, which is why I tend to put in lots of made-up details.

Nongoals. When you're building a product with a team, everybody tends to have their favorite, real or imagined pet features that they just can't live without. If you do them all, it will take infinite time and cost too much money. You have to start culling features right away, and the best way to do this is with a "nongoals" section of the spec. Things we are just *not going to do*. A nongoal might be a feature you won't have ("no telepathic user interface!") or it might be something more general ("We don't care about performance in this release. The product can be slow, as long as it works. If we have time in version 2, we'll optimize the slow bits.") These nongoals are likely to cause some debate, but it's important to get it out in the open as soon as possible. "Not gonna do it!" as George Sr. puts it.

An Overview. This is like the table of contents for your spec. It might be a simple flowchart, or it might be an extensive architectural discussion. Everybody will read this to get the big picture, then the details will make more sense.

Details, details, details. Finally you go into the details. Most people will skim this until they need to know a particular detail. When you're

designing a web-type service, a good way to do this is to give every possible screen a canonical name, and provide a chapter describing each one in utter and mind-numbing detail.

Details are the most important thing in a functional spec. You'll notice in the sample spec how I go into outrageous detail talking about all the error cases for the login page. What if the email address isn't valid? What if the password is wrong? All of these cases correspond to real code that's going to be written, but, more importantly, these cases correspond to *decisions* that somebody is going to have to make. Somebody has to decide what the policy is going to be for a forgotten password. If you don't decide, you can't write the code. The spec needs to document the decision.

Open Issues. It's OK for the first version of the spec to leave open issues. When I write a first draft, I always have lots of open issues, but I flag them (using a special style so I can search for them) and, if appropriate, discuss the alternatives. By the time the programmers start work, all of these need to be stomped out. (You might think it's OK to just let the programmers start on all the easy stuff, and you'll solve the open issues later. Bad idea. You will have enough problems resolving the *new* issues that come up when the programmers try to implement the code, without having old open issues around that you knew about in advance and could have solved then. Besides, the way you resolve anything non-trivial may have a major impact on how the code should be written.)

Side notes. While you're writing a spec, remember your various audiences: programmers, testers, marketing, tech writers, etc. As you write the spec you may think of useful factoids that will be helpful to just one of those groups. For example, I flag messages to the programmer, which usually describe some technical implementation detail, as "Technical Notes". Marketing people ignore those. Programmers devour them. My specs are often chock full of "Testing Notes," "Marketing Notes," and "Documentation Notes."

Specs Need To Stay Alive. Some programming teams adopt a "waterfall" mentality: we will design the program all at once, write a spec, print it, and throw it over the wall at the programmers and go home. All I have to say is: "Ha ha ha ha ha ha ha ha!"

This approach is why specs have such a bad reputation. A lot of people have said to me, "specs are useless, because nobody follows them, they're always out of date, and they never reflect the product."

Excuse me. Maybe *your* specs are out of date and don't reflect the product. *My* specs are updated frequently. The updating continues as the product is developed and new decisions are made. The spec always reflects our best collective understanding of how the product is going to work. The spec is only frozen when the product is code complete (that is, when all functionality is complete, but there's still testing and debugging work.)

To make people's life easier, I don't rerelease the spec daily. I usually keep an up to date version on a server somewhere where the team can use it as a reference. On occasional milestones, I print a copy of the spec with revision marks so that people don't have to reread the whole thing -- they can scan the revision marks to see what changes have been made.

Who should write the specs? Read all about it in Part 3.

Next: Painless Functional Specifications - Part 3: But... How?

Want to know more? You're reading Joel on Software, stuffed with years and years of completely raving mad articles about software development, managing software teams, designing user interfaces, running successful software companies, and rubber duckies.

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