Reviving the Lost Craft of Writing Specifications

Nat Papovich, npapovich@webapper.com
April 24, 2008

> "It seems that specs are like flossing: everybody knows they should be writing them, but nobody does. "
> -Joel Spolsky, Painless Functional Specifications - Part 1: Why Bother?

Good spec docs are security blankets for developers, risk-mitigators for management, and safe havens for business analysts. They create a common bond and communication superhighway among team members and, when correctly positioned, fit in with any development methodology. They provide a ruler with which the success of a software project may be measured. Written in layman's English, they are read by nearly everyone and snippets of them are extracted for use in other parts of the project.

Table of Contents

If all this is true, why doesn't everyone write spec docs for every project? I think it's mostly because folks don't understand the power of a properly employed spec doc. They also may have had a bad experience with a bad spec doc in a bad development process. Or, more likely, the size of their projects has grown little by little. For the smallest projects, on which we cut our teeth, a spec doc is unhelpful. But as projects grow, the use of spec docs becomes helpful and eventually necessary. If you're at "helpful" or "necessary", I'd like to help you get one written and integrated into your next software project.

Part One: What is a Spec Doc?

A "spec doc" is shorthand for "functional specification document", which is too long to say, and most people know what we mean when we say "spec doc". A spec doc describes the what, not the how. It describes what a to-be-developed piece of software will do. It describes what its users will accomplish by using the software. It describes the various features, functionality and whiz-bangs that make up the software. For the purposes of this discussion, "software" means web applications, not binary-compiled, installed applications, although many of my points hold true for that type of software too.

Spec docs always talk about *what* and never *how*. The "hows" are in a technical specification or a system design: database schemes, application frameworks, coding conventions, build processes, validation routines, and object or component models. A customer or end user can never read a tech spec and understand it, but programmers *better* be able to understand it. And although spec docs should be readable by a non-technical audience, most of my spec

docs aren't devoid of technical details and may contain information applicable to the technologies to be used - such as the choice between Flex or Ajax - and they all describe the fundamental technologies used like mobile phones or Google Gears. A spec doc describes what the solution needs to do in order to be considered a success. It describes the screens, menus, navigation, forms and business rules.

What About Graphical Prototypes?

> "If I can't picture it, I can't understand it." - Attributed to Albert Einstein

A spec doc isn't standalone - it is one-half of a delivery. My spec docs are always accompanied by a graphical prototype or something visual, showing the user interface to the application - those screens, menus, navigation and forms. Since a picture is worth a thousand words, I suppose I *could* skip the visuals and write a few-ten thousand more words, but that's a bad idea. On second thought, no I can't skip the visuals. How can I describe any work of art in words? How much effort would it take to describe an abstract Jackson Pollack painting? What about impressionist Claude Monet? Why bother trying? We're not making capital-A Art here, just an interface, but the point is the same. Include the visuals and skip unnecessary writing.

It's a commonly-recited adage that "the interface *is* the application". Customers see, use, and ultimately pay for an interface, and they only rarely read the spec doc. Prototypes and wireframes are those thousand-word pictures, but there is a lot more to an application than the user interface. (Like all that coding and database and other "technical stuff"; don't worry your pretty little head over it.) So despite the importance of prototypes and wireframes, I also use words to describe the picture as well as the stuff *behind* the picture.

Are There Any Dangers in Early Prototyping?

There are a few arguments against spec docs that I'll refute in a moment, but one drawback occurs when showing the prototypes to a customer and they say, "Hey this looks great - how long until it actually works?"

This occurs because, as I just mentioned, the interface *is* the application to many customers. When I show a one-hundred-percent complete prototype with lots of nicely-faked data, finalized graphical treatments and with working navigation, some poor sucker will think it's just a simple matter to "hook it up to a database". If my four year old daughter grows up to do programming, I hope to hell that we've pushed development sufficiently far that she'll be able to reply - without a hint of derision in her voice - that hooking up interfaces to business logic is a simple matter, I'll have it done in a couple hours. But until then, we the programmers have to actually get dirt under our fingernails and write the back-end code.

Well then, should we build prototypes that show business rules and workflows without too many of the details? What happens if I have an application that is 95% code complete, but consists of nothing but `<H1>` and `<BR>` and `<INPUT>` tags on a plain #FFFFFF background? My customer is going to think it is the worst application ever, it sucks, I'm fired. They'll wonder how I thought I could rip them off so badly. "But wait," I say, "my designer is an hour away from putting the finishing touches on the CSS which, when dropped into the XHTML-compliant markup, will turn it into a graphical work of art." Can a customer separate the two? Can a customer differentiate between user interaction and layout? No, never. Or at least, rarely: If you have a well-tuned sixth sense about your customers, then you can try blurring the line between user interaction and layout.

So I'm always careful to manage my customer's expectations. If a finalized CSS will be ready tomorrow, I don't do a plain-text review today. Sometimes I try to educate customers on the difference between the user interface and the backend code, about how they are separate layers, each independent of each other (which is a good thing).

Briefly, What Does a Spec Doc Look Like?

Shoot, all this talk about spec docs and I've neglected to describe what one looks like. My spec docs consist of five sections.

First off, I write a couple paragraphs of an overview. This is really helpful for new people joining the project and I always write it with them in mind. I never make any assumptions when writing the overview and strive to provide a nice thirty-thousand foot view that the PR people can lift.

There's not much harm in writing more scenarios but the purpose of these scenarios is to make sure no major functionality is left out and to answer questions about how a user does something using the software. There is definitely a "sweet spot" in this effort.

Secondly, I include some user scenarios. An example user scenario is something like "Purchase the items in your shopping cart," which would come after "Find a product with a known manufacturer and price range," which might come alongside "View the details of a featured product." Each of these scenarios gets a couple paragraph write up, starting either from a previous scenario's ending or the landing screen of the application. Each scenario names specific screens which are described in detail later in the spec doc. I always write one scenario for each important user interaction with the application. What is considered "important" is up for debate, but for a typical, relatively simple e-commerce example, I might have a half-dozen or more scenarios. There's no harm in writing more scenarios but the purpose of these scenarios is to make sure no major functionality is left out and to answer questions about how a user does something using the software. There is definitely a "sweet spot" in this effort. Too much writing is wasted and not enough is hazardous. I'll talk more about that later.

The next section, "Screen-by-Screen Specification" is the meat of the spec doc. Every single screen (and all functional parts of the layout) gets a description which, for a content screen, includes such information as where the content comes from, its dynamic nature and any cross-linking to other data. If the screen is a form (which most are, in web applications), the description includes a list of all form fields, their attributes (such as required, defaults, control types) and the form action or exit points. Everything that an HTML developer needs to implement the form is identified and written in the description for every form in the application. But I want all my words to count - it's that "sweet spot" again.

I call the next section "Behind-the-Scenes Specification" to contrast with the previous section. Here, I write down all the business rules, how modules interact and anything that is important to the development of the application that hasn't already been written. It may include performance metrics and technology choices. In some ways, it's a catch-all, but depending on the complexity of the software, it may be as many pages as the Screen-by-Screen Specification.

Lastly, I always have a section for notes about stuff to put in future versions. Throughout the project I always accumulate "wish list" stuff from the customer. They go in this section. In the early stages of a project, we're all just brainstorming, and what comes out of this period might not get implemented until a post-launch release or even much later. All these ideas need to be documented in one place.

That's it - five major sections. I've seen some spec doc templates include sections like a bibliography, glossary, discussions of the development methodology, and an inline revision history, all of which I don't think are important and can lead to the tragic fate suffered by many spec docs of being left unread. There is also one more thing I scatter throughout the various sections: "technical side notes". Whenever I encounter a description that would benefit from some pretty geeky details, I put a footnote or sidebar into the text which lets the casual customer-reader ignore the boring tech stuff, but eliminates all ambiguity for the serious developer-reader. I've included a sample template of my spec doc in the appendix.

What is the Difference Between a Spec Doc and a SOW?

Some of you may have written, read, or ignored a "Statement of Work" or "Scope of Work". I generally create a similar document called a "Preliminary Project Plan". When written, this document is always produced before the spec doc and it helps shape the questions that are asked to produce the spec doc. In most cases, this document is the "kick off" for the project. It gets the basic scope of the software figured out. It will set the tone of the project and begin to manage expectations and initiate the communication styles of the project. Where a spec doc describes what the application is supposed to do, a scope document is good for describing what the application doesn't do, since at such an early stage of project, everyone involved has different ideas of what the application is supposed to do. It serves to eliminate a lot of questions for project team members, and there is no excuse for a customer not reading it. Also, since a document like this is occasionally a single page, it serves to quite literally get everyone on the same page. The spec doc can be a dozen, fifty, a hundred pages.

Part Two: Who Should Read a Spec Doc?

In exceedingly rare instances, a customer will actually read the spec doc and provide valuable feedback.

For who is the spec doc written? The programmers and only the programmers. That is the intended audience, but sometimes I get a customer perusing, skimming, or claiming to have read the doc. In exceedingly rare instances, a customer will actually read the spec doc and provide valuable feedback.

In addition to "only the programmers", the spec doc is also written for other people who are intimately involved in the data or the business logic. This might be a business analyst type of person at the customer's office, or a third-party vendor. Some customers really love spec docs, but most don't. Regardless, I always write a spec doc because someone, at some point, is going to write some code for the application and the spec doc is needed for that exercise. And every now and then, a spec doc is written for purely political purposes.

As an example, imagine that you're leading ninety men on a self-supported mission for six months across an uncharted ocean to the Orient and you have to convince the royal courts of Spain and Portugal to fund your expedition. More than likely, you are going to hire a bunch a guys with compasses and astrolabes and fantasy maps to plot out your journey and tell lies about where all the gold is. On the flip side, if you're just a couple dudes in a row boat, you're going to sail into the sunset until you run aground. Sometimes, advance planning is an absolute necessity due to purely political reasons. They may not have all the answers, but they're a requirement

nonetheless.

So in general, the primary audience for the spec doc is the programmers. I write with them in mind.

Part Three: Why Write a Spec Doc?

Since the spec doc is for the programmers, what is the purpose? The purpose is to reduce questions during development and reduce costly programming mistakes. Those are the two most important jobs of a spec doc. All the other benefits pale in comparison. But notice that I said "reduce", not "eliminate". Eliminating all questions and all mistakes is too costly. It is possible to have too much documentation, i.e. too thorough of a spec doc. I try to write as much as it takes to get into that "sweet spot" of not too little, not too much. I want the coders to devour every word I wrote, not skip a single line, and not need any more information. If I wrote too much, it's a problem because it means I've wasted time and momentum. If I wrote too little, mistakes may appear in the final product and the coding may take a lot longer.

How do Spec Docs Create Efficiencies?

Spec docs are written in a human language. In the beginning of a project, it only takes a few minutes to think about a change in a feature, update the spec doc to reflect it, and voilà - the design is improved with a minimum of hassle. Making the same change directly in the code would be much more expensive. So expensive in fact that if I were to tell the customer just how expensive, they won't want it done. That would be a great way to do business if my goal is to avoid giving customers what they want. Making changes late in the game can be very expensive, no matter if there is a spec doc or not, but without a spec doc, I would be putting off making all those decisions. It is for this reason that spec docs create efficiencies by writing the right code once.

Efficiencies are also realized by avoiding death marches. This hearkens to the way a spec doc nails down the scope. When I have a scope, I can create tasks, to which I can assign time estimates. Only with these estimates can I effectively schedule milestones in my project. Accurately-scheduled milestones eliminate death marches (that is, unless I intentionally schedule death marches). More on scope in a minute.

How do Spec Docs Facilitate Team Communication?

By writing spec docs, we document and quantify all communication with every member of the project team including programmers, front-end developers, and customers. The trouble is that each of these team members prefers different communication styles. Instead of customizing multiple documents for different audiences, we create one document - the spec doc, and include visual wireframes or prototypes. This combo is understandable by all team members. I could write down all the information three or four different ways, once for each person, or I could write it all once. I save time communicating via this single point of communication, rather than continually answering the same questions again and again. The Internet slang phrase "RTFM" is popular because it's useful. It lets people who have already

The Internet slang phrase "RTFM" is popular because it's useful. It lets people who have already RTFM to get on with doing the work rather than continually answering questions which are in the FM already.

RTFM to get on with doing the work rather than continually answering questions which are in the FM already. With one document, the marketing people use it to write their fluffy vaporware ads, the QA people use it to create their acceptance tests, the business development people use it to reposition the company to destroy the competition and the developers read it to know what the heck the software is actually supposed to do.

If, for some reason, I've managed to avoid writing a spec doc, I'll still have all these people involved in the project and they're all constantly asking questions. The communication still happens anyway but it's all just ad hoc. No one is responsible for the consistency of decisions. No one knows the ramifications of one decision to others. Worse still are the tangible results. QA people "figure out" some tests to run on the application. When something goes wrong, they're not even sure if it is, in fact, "wrong". Who's to say that when you click "Submit" it shouldn't in fact, throw a 500 Server Error? QA just assumes that it shouldn't do such a thing, and you know what they say about assuming. Or maybe a particularly bold QA person hits up a programmer to ask them if they actually meant for a 500 error to come up when a user hits Submit. Because of that interruption, a programmer is off-track from what they were just doing. Maybe they'll go back and fix that bug, but when they're done, they'll sit for a couple minutes wondering, "What was I doing before I got interrupted?" Enough of these occurrences and we've officially blown the schedule.

If there are enough people foolish enough to frequently interrupt, eventually the programmers complain that they "can never get any work done around here and could you please just keep those people away from me while I work and I could get so much more work done at home or at the coffee shop, so stop wasting my time." This is a tragic response because in those cases, any user manuals or QA processes only document or test the painfully obvious things. Like the login screen only tests to see if the user was able to log in or not. Not having a spec doc means that the QA people won't test what happens if the user is on Active Directory and they hit the application, or if they're on the VPN and hit it, or what a saved login cookie is supposed to do to the login. Not having a spec means that the tech writers developing any user manuals just end up taking screen shots of the application and saying, "The login screen lets users login by entering their username and password and hitting Submit." Whoop-dee-doo.

So without a spec doc, communication between team members is harder work than with a spec doc.

How do Spec Docs Define the Scope?

Spec docs define the scope of the project, and thus with a little bit of inference, the schedule. When I know everything the application will do when it's done, I can work backwards to develop a list of tasks that must be completed. With that list of tasks, I create a larger list of smaller tasks. I keep doing that until I can accurately guesstimate the amount of time required to complete the sub-sub-tasks, assign the time to my available coders and presto! - a schedule emerges. Without a spec doc, there is simply no chance of hitting any particular deadline because I would be winging it. Remember, I'm talking about bigger projects here. Small projects get less benefit from a spec doc and have fewer risks to forgoing a spec doc.

This segues into the question of what to do when the customer needs an "estimate" or requires a "fixed bid"? I think I just answered that one - you write a thorough spec doc, break it down into tasks and provide an estimate based on your anticipated schedule. Submitting a bid without a spec doc in hand is, frankly, suicide. If you submit a bid at, let's imagine, $1000 for a project, and oh-my-gosh, you actually come in at that budget, you weren't right. You were wrong in your estimate since you didn't know what you were estimating and just happened to come in at the same amount for the development. It's like back in math class when you had to write out your solution on tests

because the teacher wanted to know you did the problem right, not just guessed at the answer. It's the process of breaking out a spec doc into tasks that lets you come close in your estimates. (Estimating is another gigantic topic in its own right, and I'm not going to get into it here.)

There is one little snafu with this plan. Many customers want that fixed bid before they're willing to drop any coin. They won't accept a ballpark range/time-and-materials budget for the development of the spec doc without any idea how much their total outlay is going to be, including the coding. This is a total catch-22. I can't commit to a project cost until I know what the scope is, but the customer won't let me determine the scope until I commit to a project cost.

This is a total catch-22. I can't commit to a project cost until I know what the scope is, but the customer won't let me determine the scope until I commit to a project cost.

What About Fixed Bids?

Sometimes in the "real world", I can't always convince a customer that building software is similar to building a house in that they need blueprints and design plans before they can get a bid on the construction. What do I do in this situation? Quite simply, I *must* determine the scope before committing to a price. Anything else is a guess which is nothing I would commit to. Here are some tricks I try:

1. Walk away. There are other customers out there, waiting for me, who know that demanding a fixed price without a spec doc is foolish. These are the customers who have been burned in the past.

2. Hire a fancy salesman in a $1000 suit to smooth-talk the customer into waiting for a fixed bid until a "discovery phase" is complete. Don't call it a "spec doc", call it "discovery". Maybe I eat the time spent on the spec doc and jack up the prices on the coding phase to cover my losses.

3. Offer the customer a fixed bid based on *just* the spec doc process. When your car is broken, you take it to the mechanic for a flat-rate diagnostic fee, right? In order to swing this solution, I craft the customer relationship into an ongoing series of agreements to get from one phase to the next. At the end of every phase, the customer can pull the plug and still have value with the deliverables provided by the previous phases. After all, the final software is just the deliverable from the coding phase, right?

4. Insist that we never work on fixed bids. This is more of a meta-solution. Since we never entertain the idea of working on a fixed bid, we never have to figure out what to do about a customer who needs one.

5. Employ the used-car-salesman approach. When a customer walks onto the lot, a good used-car salesman will sell them something, *anything*. He takes their money and makes sure they drive off the lot. If the customer must have a total price firmly fixed before we do any work, then I can certainly guarantee that *something* will be delivered within that budget. What that something is, is the scope of the project, something no one yet knows, but we get to determine it by working until the customer's budget is dried up. This is roughly what the iterative development folks do, but can create big troubles because of unrealistic customer expectations.

6. Educate the customer by encouraging them to perform their own initial discovery phase. I show them a template of a previously-completed Statement of Work or Preliminary Project Plan and have them create the first draft of it. Some customers are capable of producing this kind of document. We may end up giving away some services assisting them with creating the document, but when I have it in hand, I have a good idea of how much work it will take to perform the spec doc process.

In all these cases, what I'm trying to do is nail down the scope of the application. By determining the scope, I answer many of the big, hard questions and provide a direction and some details to the project. This is the only way to produce a fixed bid without wildly guessing.

How do Spec Docs Fit into FLiP?

If you're a Fusebox Lifecycle Process follower, you may be wondering where spec docs fit into FLiP. Since the prototype is the wireframe/front-end phase of a FLiP project, and we're just writing a spec doc alongside the wireframing phase, it's a match made in heaven. The conclusion of the prototyping phase in FLiP would include the spec doc as a deliverable. However, since the spec doc is a living document, it would be continually updated during the architecture and coding phases. I frequently use spec docs in a modified-FLiP development methodology.

Are Spec Docs a Silver Bullet?

If a development process is broken and hasn't been including spec docs, adding them in won't fix things. In fact, that addition could make things worse since the spec doc will be a new target of frustration. Completely broken project processes are most often the result of poor communication between development team and customer. And since a spec doc is the formalization of communication, it won't help the situation if there isn't good developer-customer communication. My only suggestion if you find yourself in that unenviable position is to change around the development team or fire the customer. If the project process is good but there is a new, bigger project or new, bigger customer, then a spec doc will just make the process better.

Hopefully at this point, you're on board with all the benefits of writing spec docs and convincing team members to use spec docs. But what about the naysayers?

Part Four: Why Not Write a Spec Doc?

People who make a living trashing the "formal" functional specification document creation process constantly rely on the argument that a customer cannot say what they want until they see it. These people argue that spec docs are worthless. I agree, but with the caveat that they are *nearly* worthless for *most* customers. Some customers love them and can understand them and can grok their future software when it exists in the form of a spec doc. But most customers can't get past the first couple pages of a spec doc. Remember that the spec doc isn't for them! It's for the programmers. And most programmers (myself included) dearly love a nice, clearly-written description of everything we are supposed to code.

In the process of instituting a spec doc into your development methodology, you may encounter some detractors with some of the following arguments. Here's how I answer those people.

"We do heavy prototyping."

"Agile" software methodology folks say we should skip the spec doc and work on the prototype right away. (Agilists define "prototypes" as "version 1" of the application, not the graphical representation of the user interface like we do.) But that process doesn't work for anything other than the simplest applications. What about the "if" questions? If a user does *X*, what happens? When the data is *Y*, what happens? Prototypes cannot address that. Now admittedly, we could create *so many* permutations of the prototypes that every conceivable "if" question is covered, just in the prototype, without a spec doc, but that is futile - you're better off just writing all the code. Joel Spolsky put a real-world face on this conundrum nicely:

> "Indeed, as you work on your Excel clone, you'll discover all kinds of subtle details about date handling. When does Excel convert numbers to dates? How does the formatting work? Why is 1/31 interpreted as January 31 of this year, while 1/50 is interpreted as January 1st, 1950? All of these subtle bits of behavior cannot be fully documented without *writing a document that has the same amount of information as the Excel source code*." (Emphasis added.) -Joel Spolsky, Why are the Microsoft Office file formats so complicated?

So the bottom line is that heavy prototyping can only replace a spec doc when so many dynamic possibilities are covered that the heavy prototypes are indistinguishable from the working application. This is a major, major point.

"Agile methods successfully adapt. Spec docs unsuccessfully try to predict."

Extreme Programming/Agile advocate Martin Fowler, states:

> "Agile methods are adaptive rather than predictive. Engineering methods [which include spec docs] tend to try to plan out a large part of the software process in great detail for a long span of time, this works well until things change. So their nature is to resist change. The agile methods, however, welcome change." -Martin Fowler, The New Methodology

But Fowler threw out a red herring there. Just because his Agile methods are adaptive and traditional "engineering methods" - which strongly advocate a thorough spec doc - are predictive does not mean that spec docs cannot be used in an adaptive software process. Why would the presence of a spec doc "resist change"? There is no correlation between being predictive and resisting change. A project lead who writes detailed spec docs welcomes change just like the Agilists do. The only ounce of truth in his argument is that a spec doc is another artifact that must be maintained during changes, which takes some discipline. (More on change management later.)

"The web allows us to build it/fix it/build it/fix it."

Since we're all delivering applications over the web (or at least server-side code), one response to the question of "Do we need specs?" is to say that we don't because we can just re-release whenever we notice a problem. This argument continues by saying that the "old" model of software development - the Big Design Up Front model -

was created to overcome the problem of an "installed user-base" with shrink-wrapped compiled binaries. And with web delivery, any bug fix is about 20 seconds away from making it to all our users. These people believe that we must get something - anything - to users *now*. This is their argument.

If you *can* get away with that philosophy - if you can sell your customers on that model of software development and those theories about how to create success - then more power to you. I cannot.

If you *can't* get away with that philosophy, or don't even want to try, it may be because your projects are of a certain scale, or a certain nature. Many applications cannot launch with any deficiencies. What if you're doing a version 2 rewrite of a version 1 application and, because you're in the "iterative development" camp, your new version has a third of the total functionality that the version 1 application had on launch day? By golly, everyone is going to keep using version 1, that's what will happen. In that case, iterative development isn't going to get you anywhere. (I'm not discounting the sprints in Scrum, if you're familiar with that.) There is a minimum level of functionality that you must produce in order to have the application be considered a success. Anything less - even 1% less - will be considered a complete failure. I'm not making this up - it happens. In fact, the very last project I worked on was this way. Why would existing users switch to something new if it didn't at least do everything the old version does? What if the consequences of going live without all the needed functionality are too high?

"Spec docs require unnecessary thinking."

David Heinemeier Hansson, the creator of Ruby on Rails says:

> "eXtreme Programming have [sic] a few good sayings about that: "Do The Simplest Thing That Could Possibly Work" and "You're Not Gonna Need It". Trying to guess what the evolution of the system is going to bring is inefficient at best and utterly destructive at worst. Carrying around baggage of fortune tales of one possible future is going to harm your ability to deal with the present. Worry about today today and leave tomorrow's troubles for tomorrow." -David Heinemeier Hansson, commenting on Getting Real, Step 1: No Functional Spec

That may work if you're building software for yourself, or for your own company (like David is at 37 Signals). But how many times have you "upgraded" some software or "added features" or, at worse, "overhauled" an existing application to make it do something the original developers never intended? I've been in this business for a decade, and I've had my share of these moments. Although occasionally unavoidable, these situations are exacerbated by poor coding and architecture, which aren't related to spec docs, but are related to David's comments. He's mixing up system architecture with spec docs. His concern that you will overbuild the application now and sacrifice present-day needs is eliminated if all you're doing is making sure you're not burning any bridges when writing the spec doc. Countless times in development projects, the programmers make small, daily decisions based on theories of what the future holds. "Should I write this JavaScript inline or as a function?" "Should I put this style in the div tag or add a new class to the style sheet?" "Should I perform some uniqueness verification on new email addresses to ensure we don't get duplicates?" "Should I fully normalize this table or can I get away with a list value?" All of those questions are instances of a good developer's innate ability to plan for the future. After all, that's what we do - we write code that we feel confident will work a certain way in the future. Can't we get some insight from the spec doc about how far into the future and in which direction to look?

"Spec docs aren't flexible enough. They're set in stone!"

Spec docs reduce questions and prototypes show the customer what they're going to get. But spec docs are *not* good as a contractual obligation to deliver *feature X* on *date Y*, set it stone, signed in blood. You can try to get a customer to "sign off" on something-or-other, but for most business applications, you need adaptability and flexibility. One minute a customer has a computer process to manage frozen tuna deliveries from the Pacific Rim to its headquarters in Kansas, and the next minute, they want to import Angolan sea turtles to their branch office in Dubai. I have to be able to adapt to that change by being *agile*. (No, not the capital-A Agile like all the books sell.)

The point is, I try to avoid setting anything in stone at any point in the project. At any time, I need to be able to add, remove or modify anything in the spec doc and prototypes. That requires good change management. How does the customer communicate with me? Are there regular meetings with all involved parties to refine the application's requirements and the business's needs? How quickly and by what process do I update the spec doc with changes? How do I distribute it to everyone on the project? How do I make coding and architectural changes to an application when the developers are knee-deep in code? Anti-spec doc zealots claim that rigid plans are doomed to failure and that adaptive, responsive plans are the only way to fantastically succeed. I couldn't agree more! But they consider spec docs rigid plans that bind all the parties, which is setting up a strawman argument. The question here is spec docs, not rigid plans. How you go about modifying and adapting on the fly is *change management*, which I'll get into later.

That should take care of any arguments you might encounter from people not wanting to use spec docs. But sometimes you really *shouldn't* write a spec doc. On the other hand, sometimes you really, *absolutely should* write one.

Should I Always Write a Spec Doc? Should I Never Write a Spec Doc?

Should you *always* write a spec? Should you *never* write a spec? Here is a list of questions to ask yourself when pondering that question. If you have more "Yes" than "No", then a spec doc is probably worth your time.

Questions to help determine if you should write a spec doc	Should you write a spec doc?
Is the application for yourself?	No, skip the spec doc.
Is the application for pay?	Yes, write a spec doc.
Does the project have a defined budget?	Yes, you need to define the scope.
Does the customer lack critical thinking skills?	No, they might not be able to understand decision implications.
Are you the one who is going to code it?	No, you may be able to keep it all in your head.
Are you outsourcing any of the coding?	Yes, otherwise how else will you tell developers what to write?

Is the customer's corporate culture conducive to "requirements gathering"?	Yes, they'll appreciate the time spent on the spec doc.
Is the budget micro-small to begin with, but can increase as pieces are delivered?	No, adopt a completely iterative methodology instead of a spec doc.
Is there already a relatively detailed Scope of Work or Proposal?	Yes, the customer is amenable to more discovery.
Have you ever written a spec doc before?	No, it might not make your project any better.
Do you want to write a spec doc for this project?	Yes, you have to start some time!
Is the organization high on internal politics?	Yes, companies with lots of politics tend to enjoy paperwork.
Is the project on a fixed bid?	Yes, you must define the features and scope.
Are you using contract developers?	Yes, you need to plug coder resources into defined tasks.
Is it about two person-weeks of work or less?	No, the benefits are small at best.

So there it is - reasons to write a spec doc and reasons not to write a spec doc. As you can see it's a complicated decision to make and it is highly dependent on the project, the customer, and the team members.

Part Five: How Do I Write a Spec Doc?

I hope that by this point, you're on board with writing spec docs or requiring that spec docs be a part of your development process. But hoping or mandating doesn't actually get anything done. And writing a spec doc can be a bit daunting. Where do you start?

That first shovelful of dirt is usually the hardest for me, but someone, at some point, has to actually start the project.

A couple years back, I visited the Hoover Dam outside of Las Vegas. Even if you don't have a thing for dams and rivers, it's still a great tourist spot and pretty awe-inspiring, especially when you consider that individual people made it - folks just like you and me. At one point in the tour, I watched a portion of the PBS documentary "Hoover Dam" about the beginning of the construction and there was an old-timer recounting how the hydro engineers diverted the mighty Colorado River (it was mighty back then) through the canyon to dry up the dam site. After the water flow was re-routed, the construction guys had a nice clear river bed to begin digging, building forms, pouring concrete, installing turbines, and all that stuff. This old guy was imagining being one of the first people down in the dry riverbed, surrounded by hundreds of vertical feet of rock, deep in the narrow canyon, feeling dwarfed by the years of building ahead of them, but just picking up a shovel and saying "I guess I'll start digging right here." That first shovelful of dirt is usually the hardest for me, but someone, at some point, has to actually start the project.

Who Authors the Spec Doc?

It is crucial to remember that the spec doc author isn't always the top of the food chain. In fact, in some organizations, the spec doc author is at the *bottom* of the food chain. No one reports to the spec doc author and that person's only job is to aggregate everyone else's thoughts and comments into the spec doc. If an author in that position produces a unique thought of his own, he has to submit it to the customer just like everyone else would. If, during this aggregation of ideas into the spec doc, he notices something contradictory, he can't just make a decision as to which idea is right and which should be left out. Instead, he has to take this discrepancy to the project managers, tech leads, or the customer to have them solve the problem.

In most cases though, the spec doc author is the tech lead, business analyst, project manager or coding architect. And sometimes the author is all four. In any case, it's important to remember that the spec doc is an aggregation and formalization of everything that everyone has already agreed on, not a vehicle to deliver a manifesto.

How Do I Gather Requirements?

Requirements are never just lying on the sidewalk, prone on their back, waiting for someone to pick them up and drop them into a sack. Requirements don't lay down and play dead.

Despite the fact that we call the job "requirements gathering", requirements are never just lying on the sidewalk, prone on their back, waiting for someone to pick them up and drop them into a sack. Requirements don't lay down and play dead. They're hard as hell to catch and many times, customers don't let them loose to run so you never see them. Instead of "requirements gathering" we should call it requirements "prying-from-their-cold-dead-fingers". The beginning of all projects are nothing but exercises in catching these requirements. Sometimes I glean requirements from early emails or paltry customer-produced documentation. In the case where requirements are just nowhere to be found, I can elicit a requirements conversation by asking stupid questions based on my limited knowledge of the subject matter. That always gets the ball rolling.

One thing I do as early as possible is start a list of screens that are going to be included in the application (and thus in the Screen-by-Screen Specification section). Each of these screens is going to need a full discovery process, with answers to questions such as: What does this screen look like? Where does the content come from, a database or hard-coded? How does a user reach this screen? Are there any other ways a user reaches this screen? How does a user exit this screen? Does this screen exist as part of a workflow? If so, what is the workflow and what are the other screens? If this screen contains a form, what are all the form fields on the screen and what are their attributes (like control type, default value, validation, database table/column match)? Creating this list should take a long time, It is, after all, the single most important and most lengthy part of the finished spec doc.

After I figure out the tip of the requirements iceberg, it's time to start writing the spec doc and the easiest place to begin is the overview section. If I'm having a hard time writing a couple paragraphs or a page of text about the project, then I know I don't have enough information to begin the rest of the spec doc (the Screen-by-Screen Specification section). The overview provides a decent litmus test. If I get stuck on the overview, I'll schedule a conference call with the customer where I'll babble along, repeating the factoids I think I know and asking probing questions, sometimes repeating obvious things, talking about the screens in my list and learning about new screens. And I'm usually not surprised by hearing different answers than I've gotten in the past. Each time a

customer recounts a particular wish or application feature, they add different adjectives or inflections or draw some new association with another requirement that I might never have heard of before. Sometimes I feel like a doofus, talking about really basic stuff that we covered in depth the previous week, but that's life. I'm learning a new business model and need to jump in head first even though I'll get wet and might end up doing a belly-flop.

What Are Subject Matter Experts?

By understanding the business, it becomes easier to create an application that won't have unforeseen problems after launch.

In order to ask all the right questions that will end up in a spec doc, writers need to do two things well: learn and communicate. The learning comes into play through one's ability to rapidly acquire a deep and wide understanding of the business domain at hand. If it's the futures trading market, I'll need to learn the difference between a pork belly and a lean hog, a short sell and a long put. If it's healthcare billing, I need to learn the difference between CPT, DME, and NCCI. If it's industrial manufacturing, I need to learn the difference between servo motors, teach pendants, PLCs and MIG welding tips. In order to get such a deep understanding of the domain, one needs to be a good communicator and get buddy-buddy with another good communicator who already is an expert of the domain in question (generally called the "subject matter expert"). When I also become a subject matter expert, I can translate the fuzzy, ambiguous ideas that a customer spouts out into a workable software design, and the whole time, poke holes in his harebrained schemes that won't work because of existing domain rules. By understanding the business, it becomes easier to create an application that won't have unforeseen problems after launch.

When Do I Start the Prototypes?

The prototypes can and should be created alongside the spec doc. I generally make prototypes in Visio with a Flex or HTML stencil. Sometimes if we're working closely with a fast designer, I'll have that person produce prototypes in actual HTML with Dreamweaver or whatever. Or if the project is to use Flex, I might be lucky enough to have a quick developer who can use the Flex Builder design view to generate actual swfs. Mostly though, I use Visio because I'm creating the prototypes and writing the spec doc by myself. Other than speed and ease, two technical benefits to creating prototypes in Visio are custom properties and annotations. If you are using Visio, I recommend that you spend some time learning about those two features. Creating advanced prototypes in Visio is a big subject that I hope to cover at a later date.

The drawback to using Visio is that I'm creating a layer to maintain. Prototypes in Visio don't actually become the front-end of the application - I can't export from Visio to create actual web pages. Sure, there are some other products that *do* let you export straight out to working HTML (such as Axure RP), but they don't support round-tripping all the way through the coding phase. That is, once an HTML developer starts modifying the exported files by adding in form actions and CSS files, I can't keep any changes made in Visio in sync with the developer's changes. That means that the Visio prototypes are only helpful until the coding phase begins, which is not too bad anyway.

The benefit to making prototypes using the final front-end technology (Flex or HTML) is that the prototype *becomes* the application, the front-end anyway, so I get to re-use the effort expended. What was the prototype early in the project becomes the application front-end later in the project. All the time spent on the prototype is

saved in the coding phase, so it is worthwhile to get as detailed as possible with the prototype. One drawback of doing it this way is that the prototype that was totally synced up to the spec doc is slowly destroyed. So in two months' time, we can't refer to the original prototype to understand a detail of the spec doc. Why not just make a copy of the prototype before turning it into working code, you say? Well that creates the same problem that Visio has - I would be creating something else to maintain. My rule of thumb is that if I'm going to be the one making the front-end, I strive to write the prototype in whatever technology I'll ultimately be using. But in most cases, I'm not the front-end developer, so I use Visio. The actual front-end developers would end up trashing my HTML or Flex code anyway because it doesn't jive with their style.

How Do I Document Everything?

We're catching the elusive requirements and getting stuff nailed down in the spec doc. But usually there are a lot of conversations and details flying around. It can be hard to keep track of everything since each discussion is in a different stage than every other one. I recommend three tactics to help document all the information swirling around during the spec doc creation process.

1. Record voice conversations. It's illegal in most states, but it's very helpful. Well it's illegal to record a voice call without telling the other party before the recording begins, so you should check up on that for your locale or just cover your butt and tell your customer that you're recording. The easiest way I've found to record voice calls is to buy a little $25 box from Radio Shack that plugs into my headset, into my handset, and into my mic-in jack on my computer. Whenever I want to record a conversation, I turn on an audio recording program such as the open-source Audacity, make the right physical plug-ins and hit "record". Audacity will spit out the recording as an MP3 which I can listen to later, taking notes from the conversation and getting things formalized into the spec doc. When in person, I use my portable MP3 player with an omnidirectional microphone. By recording conversations, I can pay attention to the speaker, follow up with intelligent questions and keep my hands free to pick my nose. Without recording calls, I'm constantly asking the customer, "Hold on while I write a note about that." And that wastes time.

2. Flag important emails for follow-up. At Webapper, we use Gmail as our corporate email so it's super-easy to hit the little star when an important idea appears in my inbox. Usually I'm not working on the exact feature in question when an email arrives, so I need to reply for a clarification later on. Every day or every couple days, I check out the Starred folder and one-by-one, process each email needing my attention. When a discussion is resolved, I'll keep the entire thread starred until the summary is in the spec doc, then it loses its star. I also make big use of labels. This process works with Outlook via Flag for Follow-Up and folders.

3. Write notes to myself in the spec doc. One major trouble with writing a spec doc is that so many pieces of functionality are inter-related and dependent. And when writing, I'll come across a mini-feature or description that needs a cross reference to something that doesn't exist, hasn't been written, or hasn't even yet been explored. In those cases, I make an entry in the spec doc, using my "TBD" style (more on that in a minute) and complete the cross-reference. In the beginning, my spec docs are chock-full of these kinds of notes, But over time, the endpoints start matching up and the spec doc has more black text than red. The programmers and other folks who keep an eye on the spec doc never seem to mind all the notes I write to myself in the spec doc because I'm careful to use a unique notation or style so that it is obviously a note to myself, and will disappear in a later version.

How Often Do I Release Spec Doc Updates?

At this point, I've got a spec doc that is shaping up. My overview section is complete, I've gotten most of the Screen-by-Screen Specification section fleshed out but still have a long way to go to finalize all the little details. This is a great time to release the first version of the spec doc.

Spec docs get released about every week during the heavy-editing time. I usually do it on Fridays, so I can slip to Monday if I wanted to get a piece finished. But that's rare since my goal with releasing new versions of the spec doc is that I'll do the release pretty much no matter what condition the spec doc is in. On Mondays, I'll set myself some goals to have completed by Friday and whether or not I get there, I'll release a spec doc update. This weekly-release schedule assumes that the spec doc creation process will go on over the course of at least a couple of weeks. But if it's just a one- or two-week anticipated spec doc phase, I still stick with weekly releases. There probably isn't enough difference between versions a couple days apart to warrant releases any closer together than a week. With every spec doc revision, I include a "change comparison document", which I'll get into in a minute.

With all these frequent releases, it's important that I don't end up writing too much. Writing too much detail is just as bad as too little.

How Much Spec'ing Do I Do?

Despite my previous animosity for some of Martin Fowler's blanket statements, he is spot on with this comment about comparing software development to construction:

> "Can you get a [blueprint] design that is capable of turning the coding into a predictable construction activity [for less-skilled laborers]? And if so, is the cost of doing this sufficiently small to make this approach worthwhile?" -Martin Fowler, The New Methodology

Thankfully, I don't employ unskilled laborers, so my documentation is not as thorough as that produced by an architect-engineer for construction. The amount of documentation, planning and spec-ing that software projects have depends greatly on the people who are actually doing the coding. If the development team is remote, inexperienced, or poor communicators, more detail is needed. If I'm doing the project for myself to be maintained by myself forever, less detail is needed. Every project falls somewhere along that continuum and I adjust the spec doc effort to match the needs of the project team.

After sketching out all the various spec doc sections (Screen-by-Screen Specification, Behind-the-Scenes Specification), I reach a point where I feel like the spec doc is big enough and I just need to determine how detailed to make the descriptions. How do I know when I'm done with the spec doc? I'm done when all the screens are fleshed out and all the "known unknowns" are known. But how detailed is the fleshing-out? How thorough are the business logic details described? I strive to make the size of the spec and the level of detail the leanest I can get away with. Of course I'm a little verbose, if you haven't noticed already, so I tend to over-write my spec docs. Remember that the spec doesn't actually create the final product, so the faster I can get over the spec process, the sooner the coding process begins. And if no one reads 20% of the spec doc but everything turns out okay, then I shouldn't have wasted the time writing that 20%. This is sweet spot in play, as I've mentioned before.

Due to this deep level of detail, spec docs usually need some lightening up from a standard dry academic paper. One easy way I do this is to make the body of the document a bit fluffy, but embed all the detailed information that I want in sidebars or other sections. For instance, when working on a Screen-by-Screen Specification page, I'll have a short block of text basically describing the page and what it does, but without revealing any details like the max length of the text fields or the default values of checkboxes or the inline scrolling dynamics of a combobox in a div. Those details are below the upper block of text under a sub-heading like "Technical Details for [screen name]". Customers ignore the lower section, marketing folks ignore the lower section, heck everyone ignores the lower section except the programmers who crave that level of detail because it relieves them from making those decisions themselves and ultimately, possibly, being told their decision was wrong. Which programmers hate.

While adding all this level of detail, I'm constantly interrupted and forced to revise and update details. Managing these changes is a bit tricky.

How Do I Manage Changes in the Requirements?

> The Waterfall method is not a legitimate option for software development, but is a critical view of an impossibly inflexible style.

One common complaint of spec docs is that they are useless because they're constantly out of date. Spec docs being out of date does happen, but is directly attributable either to laziness or to improper use of the difficult-to-make-work "Waterfall methodology". Describing laziness is unnecessary, since most of us accomplish it to one degree or another, but some of you may not be familiar with the Waterfall method. This method was named and described by a gentleman named Winston Royce, *not* as a legitimate option for software development, but as a critical view of an *impossibly inflexible style*. He derides the idea that software can be developed phase after phase without any iteration, without any changes to the products of previous phases and without any work on future phases. Unfortunately, many organizations didn't get the joke and do in fact implement the Waterfall method. The motivation for this naïvely ideal process is that requirements and changes to the requirements occur at the earliest time possible, when they are cheapest to implement. That's an admirable goal that we should strive for regardless of the rest of our development methodology.

A certain amount of appreciation must be given to the Waterfall method even though it was never conceived as a practical and effective approach. After all, it is an undisputed fact that a critical change to the fundamental requirements of the application made just when the project is about to be deployed will be expensive - very expensive. A much better time to make changes would be earlier in the project. No Extreme Programmer or Agile junkie will dispute that fact.

With that in mind, *the spec doc must be a living document*. It flows alongside the project in every phase, continually being updated and revised to reflect the current state of the specification - all the way up until launch day, and even beyond. Each day, whether during the initial "discovery" phase, or the actual spec doc writing phase, or the architecture or coding phases, I update the spec doc daily or more often. In fact, during the bulk of a software project, the spec doc is always up on my computer - I never shut down Word.

However, just because *I* am obsessed with the spec doc doesn't mean all the other people on the project team are

so obsessed. In fact, most folks really don't want the spec doc updated because that means they have to check out the new edits and pay attention to the changes to see if any affect them.

Like I said before, I average about one version each week during the critical feedback time, with fewer releases in the beginning and end. On each of these revisions, I print a change comparison document alongside it, which is basically a highlighted copy of all the changes (removals and additions) that have occurred since the last release and the current release. Word makes it very easy to do this with nice formatting and a few seconds of waiting. I think most team members just scan the change comparison document on each release and only go over the whole spec a couple times at most during the life of the project. Oh and I always recommend that no one goes off and does anything so silly as *print out the spec doc* since all that paper will just be wasted once the next version comes out.

I always recommend that no one goes off and does anything so silly as print out the spec doc since all that paper will just be wasted once the next version comes out.

I also keep both the weekly revisions and accompanying change comparison documents in a folder for future reference. Sometimes I'll axe a section but want it back next week when the customer decides something different. Keeping the archives are important, as is some kind of daily backup like SVN.

All of this means that you have to be proactive in keeping the spec doc updated. Software development actually requires some discipline, and writing the spec doc maybe more so than other phases. A spec doc gets out of date because someone let it get out of date. Of course, during the bulk of the spec doc writing, I'm never releasing a completely up-to-date version. There are always pieces missing and sections uncompleted. I just mark those areas with a big "Under Construction" red double-underlined font and can get on with releasing the new version. Eventually, those scary sections will be expanded to document the decisions made by the customer and development team and there won't be any "TBD" or "WIP" sections left.

The specific way that you implement the work in progress notes and deliver change comparisons depends on the tool used to write the spec doc.

What Tools Do I Use to Write Spec Docs?

It's probably obvious at this point, but I use Word to write spec docs. It's not something I'm proud of. In fact, every time start a new spec doc, I do a couple hours of digging into new tools, and sometimes I try out a few, but I never end up using anything other than Word. I think there are better tools than Word, but I can't speak from experience. Word is just what I'm comfortable with - it's my trusty hammer.

Some people insist on creating system documentation in a wiki. I'm not opposed to that. It would make great sense in that I could create a new wiki page for each screen in the Screen-by-Screen Specification section. Team members could comment directly in the wiki, preferably in a comments section rather than in the spec doc text itself. Remember that *one* person needs to be the spec doc author, so only that person should make updates to the actual content. And many wiki products do full change-tracking, which would facilitate rollbacks. I don't know how you could summarize updates into a change comparison document and I guess that depends on the wiki tool you're using. Subscribing to individual page updates would be painful since I think you'd have about a hundred wiki pages each being updated very very frequently at times. That would be information overload for a casual team member. Also a wiki would require team members to be proactive about regularly reviewing the wiki spec doc

whereas with Word, I send new version to the group via email.

What about XML? There is a system called DITA that is supposedly perfect for writing technical specifications. I've checked it out, and will try it sometime. DocBook looks promising, but I think the consensus it that DITA trumps it for spec docs. Heck I could write it in Google Docs if I had a particular affinity for online word processors that are short on features. (I did compose this essay on Google Docs, so there.)

> Microsoft Word is a dead-end street.

A couple things that are high on my list when evaluating authoring tools are interoperability and exportation, change tracking and linking. After writing my first couple spec docs that involved multiple developers in the coding phase of the projects, I really wanted to enable the spec doc to be more easily consumed by machines. Inside the spec doc is a wealth of information that could be repurposed into user manuals, system documentation, QA tests and digested by developers while they are coding. In almost all those cases, Word is a dead-end street. I can't (easily) dynamically and automatically suck out parts of the document for the user manual, other parts for the developer documentation and other parts for the acceptance testing using Word. Sure, I can add some keywords or creatively use fonts and styles to delineate those parts of the document, but that's basically a workaround for a Word shortcoming. What I need is the ability to live-read from the spec doc and ultimately, *push* changes from the spec doc into other places.

Imagine it is mid-morning on a Friday and a Flex developer is working on a screen with a dozen input fields. You're in a conference call with the customer when they decide they need to change one of those very same input fields from a text field to a drop-down. Such a simple change will have minor ripples throughout the application including possibly the addition of a new database table (to hold the values of the drop-down), changes to existing tables (using a foreign-key constraint rather than a varchar string), changes to the server-side data validation routines and all the SQL in and out of the database, but maybe none of that is written yet. Maybe the only thing that is being worked on first is the front-end, by that Flex developer. Wouldn't it be great to dynamically link-in portions of the spec doc, tagged by various keywords directly into that developer's IDE? I think DITA allows you to do this, I know Word does not.

Regardless of what tool is used, I'll eventually have a good-looking spec doc, and be releasing regular updates to customers. It's time to get them involved in approving all this work.

How Do I Do Customer Reviews?

When a customer wants to see if I've got everything down in the spec doc and prototypes, I do a review. This is the last stage before coding begins and is when I begin trying to discourage changes to the system due to increased costs and times. In the review, I bring up the home page of the application and proceed through various "use cases". I usually rip through these without any preparation since I wrote most of the spec doc already and am intimately familiar with the application. I'll start with the User Scenarios section of the spec doc, and then let the customer ask their own use cases. For each one, I'll show all the screens in the prototype that address that use case, and for every word that I say, every question the customer has and every answer I reply, I make sure that there is a corresponding note in the spec doc, a UI feature in the prototype, or that it's dead-simple and obvious for anyone who knows anything about programming. Nothing else should be left out. But of course there are those unknown unknowns:

> "... there are 'known knowns'; there are things we know we know. We also know there are 'known unknowns'; that is to say we know there are some things we do not know. But there are also 'unknown unknowns' - the ones we don't know we don't know." - Repeated by former U.S. Secretary of Defense Donald Rumsfeld

Good ol' Rummy, always clarifying things for us. But his point applies here: I'm not trying to get all the unknown unknowns into the spec doc. Anyway, back to customer reviews.

The customer reviews aren't really designed to review the spec doc. They're designed to review the prototype. It can be exceedingly difficult for any human to grok an entire application just based on the 50-plus page spec doc. We need the visual aspect of the prototype to understand the whole of the application. The prototype allows everyone to ask a question about the application, and with a minimum of clicks (depending on the prototyping medium), get a visual answer, confirmation that we are all in agreement about that question. Because I'm the spec doc author I am also able to relate any question to a corresponding answer in the spec doc, but it may take a while to find the right passage. More often that not, when I'm in a customer review session, I'll answer questions by going to the relevant prototype screens and orally answering the question. Sometimes I'll have a second laptop hooked up to a projector with the spec doc up on it for people to look at, but that's generally a waste of wall space. Mostly I just answer, "Yes, the spec doc has that," and if I'm not completely sure of it, I'll make myself a note to check up on the question later.

And that's all there is to it. Now you know all about how to write a spec doc, about creating revisions, about change comparison documents, prototypes, and customer reviews. I have a couple more comments about becoming a better writer.

What Writing Rules Do I Employ?

To be a good spec doc author, you don't have to be an amazing coder. In fact, you don't even have to be a *good* coder, but it does help to have a programming background or at least high technical aptitude. What's *not* negotiable is that you should have good writing skills, good listening skills and at least decent critical thinking skills. When a dyed-in-the-wool programmer writes a spec doc, their primary goal is in getting everything technically correct. But it's more important that the spec doc is *readable* than technically correct. If no one can understand the writing in a spec doc, then it will either not be read or all the technical information in it won't be verified for correctness. Readability is king.

My parents are English professors, but instead of teaching me writing rules, they said I should memorize everything in Strunk and White's The Elements of Style. If you've never seen it, it's a small little book, less that a hundred pages including the index and very readable. The first half covers grammar and usage but the second half (I'm talking like 15 pages total) is all about style. Out of those pages, there are a couple suggestions that many people remember from Strunk and White:

- Omit needless words.
- Avoid fancy words.
- Make every word tell.
- Be clear.

- Revise and rewrite.

And one I'm adding:

- Be funny.

Mandates are easier than putting them into practice, but it is important that, as spec doc authors, we try to become better writers. If we're all sucky writers in general, our creations will suck too, and we'll be giving a bad name to spec docs everywhere.

Appendix: What Does My "Template" Look Like?

Spec doc templates are pretty much evil for the same reasons that pre-packaged web sites are evil. They don't allow you to express the specifics of the project on which you're working. Nonetheless, whenever I start learning a new topic, I tried to lift as much material from other sources as possible. So use this template as a starting point, but ditch and add sections as you see fit.

See http://www.webapper.net/index.cfm/2008/6/16/Spec-Doc-Template to download the spec doc template. It's distributed as a PDF and Word document.