What Is Software Design: 13 Years Later

By Jack W. Reeves

People have occasionally asked whether I did any follow-on writing to my "What Is Software Design" article. The answer has basically been "No, not really." I want to make it clear that that is not because I forgot about it or otherwise changed my mind. Allow me to offer a bit of explanation.

When the article appeared, I hoped–actually expected–that I would get some type of rebuttal from some sort of industry "expert." I was looking forward to this since part of my reason for writing the article had been hopes of stimulating discussion within the software industry about the overall software development process. Nothing happened.

There were no letters to the editor that I know about and nothing ever sent directly to me. *C++ Journal* became defunct shortly after that issue, and I figured my article had gone to that great land fill in the sky that swallows most publications. I went on to doing other things. It wasn't until 1997 or 1998 that I got an email from Bob Martin (who had just taken over as editor of the *C++ Report*) letting me know there was a wiki page about my article on Ward Cunningham's `c2.com` web site. This was–quite literally–the first time I knew anybody had read my article (other than the people I personally gave a copy to).

I started to follow the discussions on the wiki page and occasionally on some news groups, but deliberately stayed out of them myself for several reasons: a) I was focused on certain other things at the time, b) it was pretty obvious that other people who had accepted what I was trying to say were just as qualified–maybe more so–to argue the points as I would have been (I specifically remember Michael Feathers writing), and c) last but not least, it still looked to me like there was a lot of opposition to the concept. Unfortunately, most of the arguments sounded pretty much like the ones I had been dealing with for almost 15 years by that point (remember, I had had the idea almost 10 years before I wrote the article).

I had grown tired of trying to deal with people who were totally incapable of getting past their own pre-conceptions to even consider the idea rationally. It was like trying to explain that the French speak a different language to someone who is convinced that "different language" really just means "different dialect of English". No matter what you say, they will parse your arguments against their beliefs and either dismiss you out of hand, or patronize you with their counter arguments. I had seen a number of projects where "design it in the code" worked, but even the people on such projects often refused to accept the reality. My level of cynicism about being able to improve things was very high.

It still is, but I think it is time I made some attempt to actually defend myself, rather than let other people do it. Therefore, what I am going to do is address some of the most common criticisms I have seen about "What Is Software Design?".

A. Initially, the most common criticism I would see can be summarized as "If source code is the design, then programmers are designers; but obviously they are not, therefore source code cannot be the design." Nobody states it that baldly, but when you parse what they do say, it comes down to the same thing. These are circular arguments that start with the assumption that programming/coding is a manufacturing type of activity. In logic, this is known as a "Begging the Question" fallacy. In essence, these people say "your assumption (i.e. source code is the design) contradicts my assumption (i.e. programmers are assembly workers), therefore your assumption must be wrong."

Someone might suggest that I am doing the same thing, i.e. starting with the assumption that source code is a design. I accept that–up to a point. While I will admit that a lot of the article reads like an attempt to prove that "source code is the design", that was not really what I was trying to do. The following quote is from the beginning of the article:

> "This article assumes that final source code is the real software design and then examines some of the consequences of that assumption. I may not be able to prove that this point of view is correct, but I hope to show that it does explain some of the observed facts of the software industry, ..."

I did not set out to prove that "source code is the design"; I will readily concede that what is a "design" is to some extent a matter of definition. The point of the article was to try to show how this assumption led to much better explanations of numerous observed *facts*. I am still waiting for anyone to offer better explanations based upon alternative assumptions.

B. These days, thanks in part to the rise of Extreme Programming and other Agile Methods, people are starting to accept (grudgingly) that programmers are not assembly line drones. Unfortunately, that doesn't mean they are willing to accept the concept of "the source code is the design". The arguments can be summarized by the example that is still on the wiki page:

> "As for throwing the whole design thing out, and just designing in code... Hahahahahahahahah no really Hahahahahahahahah :)"

This really makes me angry. For reasons that I do not understand, reasonably intelligent people insist upon confusing the concept of design as *process* versus design as *product*. You would think that anyone who passed high school would understand the difference between the process of writing a paper (for example) and the paper itself. Certainly, you would expect anyone with a college background to understand that there are often lots of different ways to arrive at the same solution.

Nevertheless, people keep insisting that my contention of "the source code is the design" means "don't do design, just code." I never said anything of the sort. What I did say was:

> "In software engineering, we desperately need good design at all levels. In particular, we need good top level design. The better the early design, the easier detailed design will be. Designers should use anything that helps. Structure charts, Booch diagrams, state tables, PDL, etc.—if it helps, then use it."

Today, I would phrase it differently. I would say we need good architectures (top level design), good abstractions (class design), and good implementations (low level design). I would also say something about using UML diagrams or CRC cards to explore alternatives. Nevertheless, I will not back away from the following statement:

> "We must keep in mind, however, that these tools and notations are not a software design. Eventually, we have to create the real software design, and it will be in some programming language. Therefore, we should not be afraid to code our designs as we derive them."

This is fundamental. I am not arguing that we should not "do design." However you want to approach the *process*, I simply insist that you have not completed the process until you have written *and* tested the code.

Personally, I think a person with his feet on the desk staring at the ceiling can be "doing design" just as seriously as someone playing with UML diagrams in ROSE. I have always known that you are better off if you put some real thought into what you are trying to do before actually doing it. People differ widely in what helps them think, however. Some people use pencil and paper. Others like white boards or even computer tools. Some people like to bounce ideas off of other people, others like peace and quiet. Some people feel comfortable with diagrams like UML. Others prefer CRC cards.

What approach they choose doesn't matter; until someone starts insisting that these intermediate designs should be products in their own right. It's the code that matters. If you get good code, does it really matter how it came about? If you don't get good code, does it really matter how much other garbage you made people do before they wrote the bad code?

Everybody that has been in this business any length of time has seen plenty of examples where someone obviously sat down and coded the first thing that popped into their mind. Later, when it became obvious that there were shortcomings with the approach, there was too much blood, sweat, and "skin" invested in the code to scrap it and do something better. Fine, we all know a little thought can go a long way.

On the other hand, any of us who has spent time on a *traditional* development project with its strict rules forbidding the writing of a single line of code until the "design" is completed and reviewed and approved, etc. knows you can waste a hell of a lot of time producing documents that are out of date literally days after the actual coding starts. Why bother?

You think we could find some happy medium of "enough" design effort, but not too much. There is no such thing. The only way we validate a software design is by building it and testing it. There is no silver bullet, and no "right way" to do design. Sometimes an hour, a day, or even a week spent thinking about a problem can make a big difference when the coding actually starts. Other times, 5 minutes of testing will reveal something you never would have thought about no matter how long you tried. We do the best we can under the circumstances, and then refine it.

One last comment: I also did not say that the only necessary documentation is the source code. I specifically pointed out in the article:

> "...auxiliary documentation is as important for a software project as it is for a hardware project."

The source code may be the master design document, but it is seldom the only one necessary.

B′. I cannot resist making a remark about a side issue that often comes up in discussions about Extreme Programming and Agile methods that is related to the above. This is often phrased as a question: what about the Less Able Programmer? The issue seems to be that only the very best programmers can "design" and "code" at the same time. To offset this, we must have all those intermediate design steps and products mentioned above to make up for the lack of experience and talent of the average programmer.

To me, this is like asking "what do we do about the less able physician?" I know the practice of medicine and software development are not analogous, but bear with me for a moment. An awful lot of the practice of medicine is pretty much rote (we joke about "take two aspirin and call me tomorrow"). Nevertheless, the medical profession still insists upon some pretty high standards of intelligence, education, and experience before someone is allowed to call themselves an MD. In other words, we want our doctors to know what they are doing.

In software development, questions about the less able programmer really come down to trying to substitute a process for intelligence, aptitude, and experience. Apparently, a lot of people think that if we force people to create enough UML diagrams (or whatever), have enough reviews, and otherwise follow a detailed process, that eventually they will figure out what they are doing and code it correctly. There is no evidence that such approaches have worked in the past, and I see no reason to believe they will work in the future. In fact, my own experience says that properly using tools such as UML involves a considerable level of expertise and experience in its own right.

C. Another argument I have seen questions the contention that the goal of an engineering effort is some type of documentation. Some people argue that the goal of engineering is a "product" and that real engineers often "build" things and those "things" are as much a product of engineering as any documentation.

This argument tries to sidestep the question of "What is a Software Design?" by implying a parallel between the "things" that other engineers build and what software developers create. Frankly, this is nonsense. I will concede that there are engineers who build "things" with little or no formal design documentation, and I suspect that even in those cases there is probably some design documentation (even if it is on the back of an envelope). In any case, I think we can safely say that such projects produce only one-off products and are usually done by individuals.

When an "engineering" effort starts involving more than a couple of people, or when it has a formal manufacturing phase, then documentation starts to loom larger and larger as the actual product of the engineering effort. You better believe that the engineers at Toyota or Motorola produce documentation, and we'll not even think about the engineers at Boeing or Lockheed. So, while it might be true that a lot of engineers do things besides producing design documents, anyone who calls himself an engineer knows what a design document in his field looks like, and probably produces such more often than not. Can we say the same for "software engineers"?

Incidentally, this contention regarding engineers and documentation was not mine originally, but instead something I picked up from an article in *Datamation* back in 1979. I agree with it completely however.

D. One final but fairly minor argument I have seen is that source code is still too high level to be a design. At least one critic wanted to call source code the "specification." His (or her) take was that the real design is what comes out of the compiler. In some sense this is just a matter of definition, but I still disagree with it.

The generally accepted definition is that a "specification" states the *what*, which is followed by a design document that details the *how*. While there is a certain amount of flexibility allowed of the compiler in determining the *how* of object code, there is certainly no creativity involved. And that is where I draw the line. When the document is detailed enough, complete enough, and unambiguous enough that it can be interpreted mechanistically, whether by a computer or by an assembly line worker, then you have a design document. If it still requires creative human interpretation, then you don't.

In software development, the design document is a source code listing.

###

<*continued next page*>

Essential XP: Documentation

Ron Jeffries
11/21/2001

> *"Outside of a dog, a book is man's best friend. Inside of a dog, it's too dark to read." --Groucho Marx*
>
> Outside your extreme programming project, you will probably need documentation: by all means, write it. Inside your project, there is so much verbal communication that you may need very little else.**Trust yourselves to know the difference.**

XP is designed to use face to face human communication in place of written documentation wherever possible. Effective conversation is faster and more effective than written documentation. When you bring people together, they need less paperwork.

Documenting Requirements

XP in its pure form has a customer (a business decision maker who knows what is needed and can decide priorities) who is "on site" with the team. We might argue about how difficult it is to get an on-site customer, but it doesn't change the fact that when you're in the room with people, you need not write them quite so many memos.

XP uses verbal discussion to explain to the programmers what is wanted. As we have said since the C3 project back in the late 90's, those discussions are commonly backed up with tables of values, spreadsheets, even extracts from requirements documents coming from somewhere outside the project. As we say in *Extreme Programming Installed*, page 28:

> User stories are made up of two components. The written card is the first. We recommend writing the story in just a couple of sentences on a card*and pointing to any supporting documentation.* The second component, and by far the most important, is the series of conversations that will take place between the customer and the programmers over the life of the story. Those conversations *will be captured as additional documentation that will be attached to the card*, will be acted out during Class Responsibility Collaborator (CRC) design sessions, and, better yet, as acceptance tests and application code. (Emphasis mine.)

Most importantly, as we see above, the requirements are documented in a form that is much more definitive than a mere requirements document: they are documented in the form of automated tests that verify the results of using the software.

We combine a focus on verbal communication with automated tests to communicate requirements. The result is much lower need for written requirements*within the team*

Some projects have a need to communicate requirements*outside* the team. This need is not addressed in the XP process. We're not against it ... it's just not something we have spoken about, except to say this::

> *If there is a business need for a document, the customer should request the document in the same way that she would request a feature: with a story card. The team will estimate the cost of the document, and the customer may schedule it in any iteration she wishes.*

Documenting Design

Like many other incremental methods reaching back as far as Boehm's Spiral Method, XP grows the design rather than synthesizing it. Again, because in XP the programmers are all together, there is little need to pass much paper back and forth. The system is built in two-week iterations, and features are typically built in a matter of days (because features are broken down to be tiny, not necessarily because XPers are notably smarter than anyone else).

In such an environment, design artifacts are typically ephemeral. The team may draw some UML on a whiteboard or a tablet, then sit down to build the feature.

It is common for XP teams to have some pictures of the system's design on the wall for extended periods. I observe that these seem to serve more as decoration than as documentation: people don't look at them very often. They ask each other, or they pair program with someone who knows the answer. Why do they do this instead of look at the pictures on the wall? I believe it's because verbal communication and pair programming work better. But I have no proof -- only experience.

XPers teach that if the team needs a document, be it a drawing, a table, or words in a row, they should produce the document. We also teach that if you produce a document and don't use it, that might well be a hint that you didn't need it after all.

Again, this is inside the team. If there is a need to communicate outside the team, and it can't be done by coming together, then of course it is just fine to write something. That just makes good sense. We're Extreme, not stupid.

When it is time to hand off the software, whether to maintenance people or to users (perhaps programmers who will use the software to build other things, or other kinds of end users), then of course you need to build appropriate documentation to go with it. In the case of users, I'd think this would be just like any other kind of user documentation. In the case of maintenance programmers, we typically suggest a short document describing the system's architecture or metaphor (perhaps with some UML diagrams), the key concepts, the important classes, and so on. This should, we suggest, be thought of as an introduction to the system, a high-level index if you will.

Documenting Code

XP has very high focus on incremental development. The development cycle is to begin with Simple Design, communicated through Pair Programming, supported by extensive Unit Tests, and evolved through Refactoring.

For this to work, it must be possible to refactor the code: the code must be very clean and very clear. There are rules (see elsewhere a discussion of the rules of simplicity relating to reusability) that help programmers produce clear code. In essence, if there is an idea in our head when we write the code, we require ourselves to express that idea directly in the code.

We treat the need for comments as "code smells" suggesting that the code needs improvement, because it isn't clear enough to stand alone. Our first action is to improve the code to be more clear. Whatever we cannot make clear in code, we then comment.

Unit tests are also documentation. The unit tests show how to create the objects, how to exercise the objects, and what the objects will do. This documentation, like the acceptance tests belonging to the customer, has the advantage that it is executable. The tests don't say what we think the code does: they show what the code actually does.

Inside the team, this is typically enough code documentation. Outside the team, similarly to the sections above, you might need more. Again this would be most likely at some time when the code is to be transmitted to someone else. At that time, if a document is needed, we suggest that it should be written.

Manuals

The best way to do manuals is probably to have the writers work right alongside the programmers, as part of the team. For a lighthearted look at that idea, please see Manuals in Extreme Programming.

Other Documentation

As XP is intentionally a *minimal* methodology, we do not follow the RUP path (an honorable path, just a different one) of listing all the documents you might want, from which you select those you deem suitable. Instead, XP puts the people who are stakeholders in the project together, in an environment of rapid feedback, and trusts them to work out what additinal things they need ... not just documents but any other form of project enhancements.

Some people argue that this is risky, that teams "can't be trusted" to figure out what's needed. We find that in fact teams that set up rapid feedback cycles learn quickly and do just fine.

Conclusion

I have covered the main documentation areas that came to mind. I think it should be clear that we desire strongly to minimize documentation, but that just like most everyone, we want all the documentation that the project actually needs, and no more. If there is a difference in focus, it is that we believe that with close person-to-person communication, the team will quickly find out what is needed. Therefore we specify a safe minimum of process -- including documentation -- rather than piling things on just to be sure.

If you have questions about specific documents and when we'd recommend them, email me, or post a question on the extremeprogramming group on yahoo. We might even get a followup article out of the deal.

Manuals in Extreme Programming

Ron Jeffries
11/21/2001

Here's a bit of a rant I wrote some time back, talking about how to write the manuals for an XP project by using writers as part of the team. It's a serious proposal, written with tongue a bit in cheek.

Let's think about Extreme Programming, software products, and their manuals. Some starting points:

- Everyone who has done software products knows that almost no one reads the manual. We know this from the technical support queries we get. We know it because most of us don't read the manuals either.

- Web apps don't come with manuals, and they're kicking butt. Some of them have a couple of help pages. Many have nothing but the instructions on the page and the flow of the buttons.

- More and more software today is delivered on a CD, and the only manual you get is the size of the CD box. Seems to work fine - science has found that more people read those little books - it's thought that they're looking for the lyrics.

- And hey: XP projects build the software with the highest business value first. The stuff at the end doesn't matter as much as the stuff at the beginning!

Put this all together:

- Write an easy to use product that doesn't require a big manual. Then - duh - don't write a big manual. Don't listen to the pointy-haired guy saying "but our customers expect big manuals". Your customer hates big manuals. He has shelves and boxes full of them just like you do.

- Make the product as easy to use as a web site. This will please the majority of your customers who don't read the manual anyway, and will reduce the size of the manual for those who do. Back injuries go down, customer satisfaction goes up. Stock price through the roof. Porsches all around.

- Document as you go. Plan to have documentation one iteration behind development. If you (the customer) decide to change things in some iteration, just remember that you're also deciding to change the docs in the next.

> Documenting a feature shouldn't be any more difficult than implementing it. If your programmers can invent the bloody thing from scratch in one iteration, your writers ought to be able to write it up in the same time and stay only one iteration behind. It's just reporting, after all. If the writers can't keep up, hold THEIR feet to the fire - it's their turn. Let them write *Extreme Writing - Embrace FrameMaker* in a few years if they need to.

When the product is code complete, it will be documentation complete a couple of weeks later. Two more weeks should give you the hard copy, and you can beat that if you get creative. For example,

print the document before the ship date, and do an 8.5x11 supplement for the last two iterations' stuff. Better yet, don't print it at all. Do the docs in HTML and ship it on the CD or put it on the web.

Do you still need hard copy? Well, drop ship it direct from your printer. Or save a tree: give people a coupon. Many of them won't use it, and the mail service gives you a few weeks to finalize and print the manuals. Charge extra for the manual. Increased revenue, stock price, more Porsches.

But wait! Don't answer yet: there's more!

XP programmers do Continuous Integration and release all the time. Small releases, remember? So have XP *writers* cut their documents every day, week, iteration, release. They'll get really good at it, just like the programmers.

In a software product shop, Extreme Programming will wind up implying Extreme Business. We all know that one of the reasons the software has to be done by [insert impossible date here] is because it takes X long to do the documents, Y long to do the packaging, Z long to do the shipping.

What made them think the software schedule was flexible, when none of the others were? Probably it's just that programmers are loyal enough to try harder, and dumb enough to think trying harder makes them go faster. The shipping department knows better, and the writers are more articulate.

Well, bite me. XP teams can deliver software by the clock, with very good control over what's on the disc and what's deferred. An XP team with associated writers can deliver a documented product at most one iteration after code complete. I'm willing to bet that if you let the writers be part of the team, they can do better than that. The rest of business can get on board or face the fact that they, not the programmers, are the cork in the orifice of progress.

Let the shipping department take the heat for a change. Put some good writers in the room with the programmers and let them figure it out. Don't set their goals or their process, just let them see what's up. Maybe all they'll do is work from the release at the end of the iteration. Maybe they'll find a way to start writing half-way through the iteration. Me, I bet they'll write the Extreme Writing book as a side effect.

Extreme Programming. Ship better software, on time, while killing fewer programmers and fewer trees. Humanistic, my tail. XP is GREEN! And remember - it's not easy being green!