Misunderstanding Agile Design

Now I like to take shots at myself for producing drivel now and then, but today, I'm going to take a shot at someone else's drivel. I really should be working right now, but I really need to stop a moment to respond to some FUD. Once again, Joel Spolsky sprays more ignorance on his readership with this quote...

> I cant tell you how strongly I believe in Big Design Up Front (BDUF), which the proponents of Extreme Programming consider anathema. I have consistently saved time and made better products by using BDUF and Im proud to use it, no matter what the XP fanatics claim. Theyre just wrong on this point and I cant be any clearer than that.

First, as Brad Wilson mentions, **Agile does not mean no design**.

The primary mantra of agile methodologies is to **do only what is necessary, and no more**. For a product company like Joel's FogCreek, a functional spec is absolutely necessary. (As an aside, I'm a fan of his Painless Functional Specifications Series and have used it as a template for functional specs on several projects). They are not treading new ground with their products and the requirements appear to be very stable from release to release. For example, for CoPilot, Joel dictated the requirements which the interns implemented.

However, I'd point out that the spec he published for all to see is a great example of doing what is necessary and no more. Notice he didn't list out the specific database tables nor class diagrams. This spec is not an example of big design up front. It is a great example of doing just enough design up front as necessary. How very agile of you Joel and you weren't even trying.

The second fallacy is that Joel takes his narrow product-based experience and applies it to all of software development. When you are the one who gets to define requirements and your project does not explore new ground, Big Design Up Front hands down can work. But try applying that approach to a client project and watch with horror as three months into the project, the client changes his mind on a feature and leaves you with a hunking mass of outdated and useless UML diagrams you spent eighty man-hours producing.

Agile methodologies are designed to manage change. When you don't have change to worry about, you can resort to BDUF (though even then I'd only do what is necessary). Agile methodologies weren't designed to handle developing the software for the Space Shuttle. Requirements are fixed and hardly change in such a project.

But most real world projects have a lot of change. Where does that change come from? The client! There are other sources of change during a project's lifecycle as well, such as new technologies and from new ideas gained during the project, but the majority of it comes from the client changing his or her mind.

Your typical client knows jack shit about how software is really developed. Yet you expect the client to be able to express extremely detailed requirements for what he or she wants? Might as well hand her a keyboard and tell her to write the code for what she wants. Would you try that with a home builder?

"Hey, I've written you a list of exactly how I want my house to be. I'll be back in a month to see the finished product. Can't wait!"

I sure hope you wouldn't. Most likely you'd want to check in every now and then and see how things are going. And as you see the house develop, you might change your mind about a few things.

Developing software for a client is very much like that. **A client often doesn't know what she wants until she sees it**. As the project unfolds, the client (and development team) learns more and more about the product and starts to realize that some of her initial requirements don't really make sense, while also recognizing that there are other requirements that she hadn't thought of, but your demo reminded her.

Try BDUF on a project like that, and you're setting yourself up for disappointment and failure. That's where an agile

methodology really shines. Divide the project up in iterations, do just enough up front high level design to give the system coherency, and then flesh out the design during each iteration via some up front iteration level design and refactoring. Again, do just enough design as necessary, but no more.
