Position Paper #1

What drives design?

I think that this question was pretty well answered in the *Big Ball of Mud* paper by Brian Foote and Josheph Yoder (http://www.laputan.org/mud/)! Its disheartening to me that those drivers (ignorance, finances, politics, schedules and other external factors, etc.) are not under my control, nor of people who are most interested in this workshop.

What drives me to design?

I get great pleasure in seeing systems I designed running in production, in particular those systems that:

1. meet the customer's current business requirements
2. meet the customer's evolving business requirements
3. demonstratively behave correctly
4. require little support

It can be quite challenging to design such systems and its even more challenging when the customer is not as fully committed to these qualities as I am: those are the kind of customers I try to avoid. I guess that makes me an artisan.

What drives good design?[1]

Although I have been able to articulate for some time the qualities of a good design, its only been recently that I have really begun to understand how I go about doing it. I believe that a good designer needs to:

1. Understand the real problem
    - don't let the customer distract you by talking about something off topic because they are too embarrassed to admit that they don't really know what is needed
2. Understand the problem domain
    - find the underlying principles that govern the domain
3. Layer the design and make the top layer of the design readily understandable by the business users of the system
    - strive for completeness and soundness at this layer, as well as all the underpinning layers
4. Partition the design to reduce coupling and increase cohesion
    - Eric Evan's *Domain Driven Design* provides some excellent guidance on this topic
5. Don't assume a perfect world
    - there will be errors & faults, and recovery mechanisms will be necessary
    - provide enough information so that users can do their own recovery
6. Don't forget to do a mini-business case for everything
    - its the essence of design, but often forgotten

I believe these principles are sound, but I question if they are complete.

---

1 Ironically, the call for participation in this workshop doesn't even ask "What drives good design?"! I couldn't even find the word "good", nor any synonym (other than "positive", which was always paired with "negative"), in the call. If it had, then much of the drivers in the *Big Ball of Mud* paper can be dismissed for the purpose of this work shop. Maybe I should watch Rebecca's talk again to see if she talks about "good design". If she does, then the workshop implicitly is about good design.

Position Paper #2

There is no such thing as Test Driven Design

I reluctantly was drawn into XP and TDD (that is Development, not Design). Some of the XP principles I understood and agreed with and some I did not understand, but my biggest complaint was that it has no design phase, which is my favourite part of the development process. The first time we pitched an XP approach, we won the contract. Now I was committed. That first day of coding was very tough for me: I wasn't sure that I could cope without having a design to point me in the right direction. Adding to to the unease was my reluctance to write tests and my disdain to write poor code in order to get to passing tests as quickly as possible. However, I had a lot of fun refactoring. My XP buddy often thought that I was procrastinating when I claimed there was another code smell and wanted to do some more refactoring. In fact, I could do more refactoring faster and more surely because of the tests: I could be fearless, to use the XP terminology. A couple of months into the project, when we were doing our first "big" refactoring, I realized that XP actually has quite a bit of design, its just goes under the heading/label of "refactoring", and it is not inherently part of the "testing".

Even the XP literature does not discuss TDD in terms of design: the discussions revolve around just-in-time requirements gathering, codifying the requirements and you-aren't-going-to-need-it (YAGNI) because the business has changed (or the ROI was not there). This is to say that we XP developers really do not understand the problem domain at the beginning of the project, and that we need a mechanism to get paid (for providing value) while we learn the domain. I can speak from experience of several instances where the final design of some component was significantly different from the way I had envisioned it earlier on in the project. Had I the deep domain knowledge at the beginning of the project, then it may have been more efficient to use the "waterfall methodology". However, since I have the ego of most good designers, and believe that there isn't a system that I could not design if I had enough time. The benefit of TDD is that the domain learning is much quicker and much better absorbed, and this is the real reason I practice Test Driven Development.

TDD Aside

When I started practising TDD, my XP buddy and I would have great discussions (arguments?) about how to write the tests. I never have been very fond of unit tests (the kind of tests most talked about in XP literature) because I feel that they tie the testing too close to the implementation. The big problem is that it becomes very difficult to refactor class responsibilities without simultaneously changing the tests, which significantly reduces the confidence that the code changes were a pure refactoring. To support my position, I can offer anecdotal evidence from several colleagues who have experimented with TDD and found it to be a significant hindrance because they felt they were doing twice as much work because they had to change the tests every time they wanted to change their code..

My preference would be for the tests to faithfully codify the business requirements. Ideally, a good portion of tests should be written by the business users of the system. Realistically, the best we can hope for is that the business users can read and understand the tests. It would be nice if a system specification could be generated automatically from the tests. This style of testing has since been given the name of "Behavioural Driven Development" and one of its trade-offs is a reduction in test coverage percentage.

I think that I am doing a pretty good job of TDD, but I wonder if I could do better, and I wonder if there is an ROI to apply more "design" to testing.

Position Paper #3

The sky is falling! The design is missing!

For our XP projects, I know that we have a fair number of design artifacts – mostly snapshots of freehand drawings on a whiteboard – but they are scattered mostly throughout a number of email messages. Even if I could find them, I am not sure of what use they would be to a new team member because the context would be missing. Our projects do have very high level designs that are documented on a few wiki pages (which could be read and understood in about 10 minutes), but they mostly reflect the partitioning of the domain as identified by the customer: there is no technical depth. I don't have enough experience with adding new team members to an XP project to know what artifacts would be of most value to them. I suppose if they wanted it, I could use one of the tools that reverse engineers, from the source code, the entity-relationship model or class diagrams.

Mind you, I am not sure how much value a detailed technical design is to a new project member. I have been the principal author of a number of in-depth design documents, at least one of which lead to an application that was recognized by the Smithsonian Institute for Technical Excellence. Yet, the designs were referred to rarely past the middle of the development phases, and the applications would slip out of synch with the designs. This is because there was little or no resources allocated for updating the design document when team members found alternate, possibly better designs, or when they had to deviate from the design in order to accommodate some aspect of the domain that was not well understood during the design phase. This would always lead to lots of questions from members added later in the development phase, and sometimes the best answer I had was that "working code always trumps a beautiful design".

On reflection, it seems to me that a design document is as much a project managers tool as it is a developers tool. It gets used to estimate the overall effort, to partition the effort, and to allocate resources. Project managers use the design equally as much towards the end of a project because it is the yardstick against which progress is measured. So, project managers use the design throughout development phase whereas developers tend to use it much more towards the beginning of the development phase.

That Design is No Impressionist Masterpiece

I have done many designs using primarily prose and a few using design/diagramming tools. With the prose-based approach, I have found a way to make my designs understandable by the business users, which greatly improved their confidence that we could deliver a functioning system. The problem with prose-based designs is that there are no tools (that I know of) for checking the consistency of the design. On the other hand, when I used a design tool, I had much more confidence in the consistency of the final design. The tools tended to be used on larger projects where there were several designers (not just me alone), and the resulting designs would cover an entire wall or two of a conference room. The business users never got a good understanding of these designs: we could explain small parts of the wall, but never the whole thing. I often wished that we could step back from the wall, and as we did so, that a picture of the overall design would emerge, like some pointillist masterpiece by Georges Seurat or Paul Signac. However, there was no hope because we could not get the tools to group entities and classes in any rational fashion. Hopefully the tools have improved in the 15 years since I last used them.