An alternative view of design (and planning)

Traditional software development techniques highlight the importance of planning our software through the creation of designs. We often measure our work against plans made before coding starts, and many organisations use adherence to plan as a management control mechanism. Yet just about anyone involved in software development knows that time estimates are usually wrong, and program code doesn't always follow designs produced to start with.

Many in the agile process movement openly question why we bother with plans at all. "Do the simplest thing possible" becomes the only design decision we need to make again.

I'd like to propose that planning is useful, but not necessarily for the reasons we often think it is...

Why plan?

Although the quote is sometimes attributed to others, I believe it was future US president, General Dwight D. Eisenhower who said:

> "In preparing for battle, I have always found that plans are useless, but planning is indispensable."

The sentiment isn't restricted to the battlefield, I'm sure many software developers have had recourse to this quote on occasions. What lies behind it is fact that we are not blessed with perfect future vision. Most plans contain assumptions about how the future will unfold, many of these assumptions simple extrapolate from the way things have worked in the past - or how we perceive the things to have worked. Many unknowns, and plenty of unknowables force us to make assumptions.

Even if all our assumptions turn out to be right, we have no guarantee that our plan is complete. How much detail do we need in our plan? Too little detail and you risk missing something important, too much detail and you'll never get beyond planning - sometimes called "paralysis by analysis."

Some assumptions will be conscious and may be explicitly stated, others will be implicit and undocumented. There will be many implicit assumptions in any development effort, these are derived from our existing knowledge of the technology and business and on the whole offer short-cuts to thinking. However, some of our implicit assumptions will cause problems. Planning is ones means by which we can flush out these assumptions and challenge our existing mental maps.

That plans assume foresight, and that foresight may be wrong is fairly obvious. What is less obvious is that plans also assume communication. Even the best plans can fail because they are not communicated clearly, or the receivers don't act on the information as we expect.

Such problems with planning led Arie de Geus to question the role of planning. In the traditional model planning is a tool which attempts to predict the future, the plans are then used to command and control our activities. In contrast de Geus sees planning as tool for learning:

> "So the real purpose of effective planning is not to make plans but to change the microcosm, the mental model that these decision makers carry in their heads." (de Geus, 1988)

Like Eisenhower, de Geus is suggesting that we don't make plans so we can follow them, we make plans to map out the terrain - that is, the problem domain we face. But he also goes further in suggesting that by using planning we can accelerate learning. He suggests planning is a game, a game were we can experiment with different rules and safely make mistakes. The imporant part of planning is not the output but the process.

de Geus formulated his ideas as part of the planning group at Royal Dutch/Shell, the head of this group, Pierre Wack used *scenario planning* to explore the future. Perhaps the best book on scenario planning is Peter Schwartz *The Art of the Long View,* Schwartz is clear about the role of scenario planning:

> "Scenarios are not predictions. It is simply not possible to predict the future with any certainty. ... Often, managers prefer the illusion of certainty to understanding of risks and realities. If the forecaster fails in his task, how can the manager be blamed?" (Schwartz, 1991, p.6)

How do these ideas play out in software development? Before I attempt to answer this question lets just recap on the two key ideas suggested:

- Firstly, while plans may help us to explore the future, even the best plans will not describe the future.
- Secondly, the planning process is actually a learning exercise, and it is this process which we value, not the plans we produce. The learning that occurs during the process is a result of communication, exploration and the surfacing of assumptions. Importantly, this experience is shared by the whole team.

What planning do we do?

Oranges aren't the only fruit, and project schedules aren't the plans we make. Specifications, flow charts, structure diagrams, pseudo code, UML diagrams, interaction diagrams, and a host of other diagrams all constitute plans we make in advance as a way of exploring our problem and solution domains before we start coding.

In fact, even when we start coding we still planning. Every function which is written with a stub or is flagged "TODO" is part of a plan, the more we code the more the "plan" becomes an implementation.

Planning can be a point of tension between managers and software developers. On the one hand, some managers understand progress to mean lines of code written - Steve McConnell calls this WISCA syndrome - "Why isn't Sam coding anything?" (McConnell, 1993). On the other hand, excessive planning, document writing, project schedules, and fancy architecture diagrams can act like quick drying cement to stop a project from progressing.

Sometime we do just jump in and code. Occasionally this is because the problem is so simple the solution appears obvious, or more likely, we've seen the problem before and know a solution that works. Other times the problem is so hideous that we don't

know were to start so try something. In this mode the code is part of the planning process, we're exploring the terrain by experimentation.

The value of prototyping lies in its role as a planning tool. The prototypes are written for different audiences but typically allow people to learn about the solution before committing themselves to a solution. By viewing the prototype, both developers and clients can accelerate their learning about the solution.

Test first development is another form of planning. By considering the test cases before we write any code we are again exploring the problem domain. Planning the tests gives us a chance to improve our understanding before we start coding. Almost as a side effect we get a test suite and save ourselves some time later on.

The traditional view of software design is akin to building development, the plans tell us where to build a load-bearing wall. However, with software we don't always know where the load will occur. For example, it is almost impossible to predict where the performance bottlenecks will be in a complex piece of software - the costs of "premature performance optimization" are wide accepted.

Even if building design was an accurate metaphor for software design it is not without flaws itself. Stewart Brand (1994) has criticised architects and lack of flexibility advocated some alternative ideas (see sidebar on scenario planning.)

Planning as vision formation

The activity of writing program code requires us to make design decisions with every line we write: Is a *for* loop more appropriate than *while* loop here? A *template* or a *class* there?

Of course, we could draw up more detailed plans to help us, but the more detailed our plans the more the plans are the code. (This is one of the failures of mathematical formal methods, the resulting "specification" can be more difficult to maintain than the actual code.) And at the end of the day, we don't deliver plans, we deliver working code, we want to make our design decisions at the most efficient point, sometimes this is high level, sometimes this is low level.

What we require is a framework that allows us to make all our decisions in a coherent manor. If we have some guiding vision for the system there is less need examine each decision in minute detail.

Traditionally, we would ask a System Architect to draw up a high-level design for a system. This could be refined by "designers" and implemented by software engineers. The engineers are prevented from making mistakes because the plans control what they do.

However, not only does this model assume that the architect and designers get the design right, but it also assumes the model is communicated with complete clarity and understood by everyone involved in a timely fashion.

How often do we see provisional design decisions become fixed elements of the system? By the time we realise part of our design could be better not only is there too much code to change but there is a bunch of developers who need re-educating.

For a system to remain flexible and soft, it is not only necessary to keep the software flexible but the people must be capable of change too. Thus, we return to de Geus idea that planning is part of the learning process.

(Notice I say the "people must be capable of change", not the "change the people". Often the first reaction of new developers on a software project is to claim the existing code is unmaintainable and the whole thing needs replacing.)

In the de Geus's world, everyone is part of the planning process. We plan so that we create a mental model of the system which is shared by everyone. To put it another way, by allowing everyone to participate in the design everyone will buy-into the architecture and understand how it effects them.

Ric Holt of the University of Waterloo has suggested that software architecture is most usefully thought of as a mental model shared by the development team. It is more important for the team to hold a common understanding of what is being created than it is to create highly detailed descriptions of technology. Holt's conclusion echoes Conway's Law (1968):

> "When teaching about or designing software architecture we should always remember that the architecture is intimately intertwined with the social structure of the development team." (Holt, 2001)

And so we return to team work. For software development to succeed the team needs to work together. What, you may ask, is the role of the architect here?

The role of the architect, indeed any other manager on the project is changed when we take this view of planning. They no longer sit in a darken room and emerge with a completed blue print of how the system should be. Their role becomes one of facilitator.

Architects may still sit in darkened rooms and think grand thoughts, they may still examine strange new technologies, but they no longer emerge with a plan. Instead they emerge to facilitate discussions, their research may play a part in the architecture and vision created by the team but for a team to truly buy-into a vision, and to truly understand, the architecture each team member must have a hand in creating the vision.

Emergent design

While we may like to think that the plans we make at the start of a project actually describe the system we create the reality is usually different. We find a need for objects that were never included in the object model, the algorithms described by flow charts and structure diagrams turn out to be buggy so the code is different, and refactored code quickly diverges from the plans.

As we develop at the code level a design emerges. To a greater or lessor degree this mirrors our pre-coding plans (assuming we made any). But over time the code becomes the best place to look for design. If we want a high level view of what and how a system works we are better abstracting from the working code than examining blue-prints devised before the code was written.

Acknowledging that design is an emergent, ongoing process again challenges the traditional role of design and architecture. However, when we re-perceive design as a learning process through which we create a common vision and understanding of the system, and we re-perceive the architect's role as one of facilitator rather supreme-planner then emergent design is a natural result. Because the design which emerges comes from a group of people rather than an individual the design is shared and understood by all.

What about plans as documentation?

Of course, plans have another use, they are the place we turn to first when confronted with a new system. Day one on a new job and we all expect to be given the system design, and usually we find it doesn't exist, or, at best, is out of date.

The fact that plans seldom reflect the realised system has long been known, and famously led Dave Parnas and Paul Clements to write about "A rational design process and how to fake it" (Parnas, 2001). They argue that after building our systems, we should go back and create the documentation we would have created if we had perfect foresight.

Although this may seem a novel idea it suffers from a number of problems, not least that it assumes we will be allowed time to write documents once the development has completed.

More dangerous is the fact that we are introducing an element of dishonesty into the process. No matter how well intention our motives we are doing something subversive, is it any wonder that managers ask "Shouldn't you have done that before you started?" Introducing subterfuge into the process is counter productive as it also undermines trust.

Rather than fake our plans it is far better to be honest and say "We wrote this after the event." If we want documentation for future developers than we should produce that as a specific task based on the working system.

Unfortunately there are two catches here. Firstly, much of what we learn when developing software is tacit knowledge. It may be shared by the team but it is actually incredibly difficult to write down. The fact that we can codify it at all in program code is pretty remarkable - although often we may not realise we're doing it - implicit assumptions again.

We can try and compensate here by writing copious amount of documentation. However, this brings us to the second catch which observant readers will have spotted already. Remember de Gues point about speeding up learning? The more documentation you produce the longer it is going to take new people to come up to speed on the system. Less can really mean more, less documentation can result in more time actually learning about the system.

In fact, copious documentation may make things worse still because we come to rely on words and diagrams. Assuming these are accurate (a big assumption) we have now changed the nature of the issue from one of *problem solving* to one of applying a documented solution.

However, software development is inherently a problem solving activity. If it wasn't we could automate the process. Therefore, although they may help, documentation and plans never contain the solutions, they may actually be false friends.

Final thought

One final thought, in de Geus' model of planning as learning it is the institution that learns - were we interpret institution in the broadest sense. He says:

> "And here we come to the most important aspect of institutional learning, whether it be achieved through teaching or through play as we have defined it: the institutional learning process is a process of language development. As the

> implicit knowledge of each learner becomes explicit, his or her mental model becomes a building block of the institutional model." (de Geus, 1988)

The emphasis on language creation is similar to the pattern community. By developing a language whether through patterns, planning or scenarios we create high level abstractions that allow us to discuss complex topics.

Other parallels exist with patterns, like patterns this view of planning seeks to turn implicit knowledge into explicit knowledge, both focus on creating building blocks, pattern writers and scenario planners are directed to focus on forces and particular importance is attached to naming both patterns and scenarios.

How different, and how much more exiting, to view planning this way instead of an GNATT chart.

Bibliography

Brand, S. (1994) *How Buildings Learn: What happens after they're built,* Penguin.

Conway, M. E. (1968) How do committees invent?, *Datamation*.

de Geus, A. P. (1988) Planning as learning, *Harvard Business Review,* 66, 70.

Holt, R. 2001 *Software Architecture as a Shared Mental Model*, http://plg.uwaterloo.ca/~holt/papers/sw-arch-mental-model-010823.html, Position paper to ASERC Workshop on Software Architecture

McConnell, S. (1993) *Code Complete,* Microsoft Press, Redmond, WA.

Parnas, D. L., and Clements P.C. (2001) *A rational design process: How and why to fake it* In *Software Fundamentals: collected papers of David L. Parnas*(Ed, Hoffman, D. M. a. W., D.M.) Addison-Wesley.

Schwartz, P. (1991) *The art of the long view,* Bantam Doubleday Dell, New York.