



## **Tyranny of the Metaphor**

### The Slippery Slope of Scheduling Software

Software is not a house.

It seems almost everything in the software world is a metaphor. I sit here typing in a “page” of a “document” that will become a “file” in a “folder”. In software, we create and use metaphors every day, almost without thinking about it. Metaphor creation is a natural consequence of working in a virtual world. Humans often need to map complex concepts to something tangible in order to gain understanding. Unfortunately, this metaphor mania extends to management expectations on the software development process. Management often wants, expects, or worse yet – tries to create project schedules for software development using the same techniques we might expect a contractor to use on a home construction project.

It seems reasonable enough on the surface. There is a pile of work to be done, and one of the project manager’s duties is to analyze that pile into something more manageable – usually a schedule with a resource plan and budget. The well-meaning project manager looks to the wisdom of conventional project planning for guidance in calming the chaos. After all, project management is not a new science. For decades, (if not millennia), techniques for taming the complexity of large collaborative projects such as construction have been well understood. If we just pretend our embedded software project is a house, we can organize, estimate, and schedule the tasks involved in the same way our contractor would, and we’ll be off and on our way to on-time, on-budget project completion and that big promotion and bonus.

Did I mention that software is not a house?

If we were a contractor building a house, we’d have a specification (a set of blueprints) before we began. This specification would be the definitive description of our completed project. Of course, we could expect changes and variations along the way, but the specification would give us a clear enough

idea of what we were building that we could create a reasonably accurate development plan.

However, software IS a specification. Our software is a human readable (OK, that's debatable) representation of the exact functionality we want. Our compiler will translate it into a working executable – our "house". In the pathological case, if the specification is complete to that level, development time is highly predictable. It corresponds to the runtime of our compiler. That would make our project 99.9% planning and 0.1% execution.

Yes, I hear that collective sigh. Of course it is possible to create high-level specifications for a software project. You simply have to choose your level of abstraction and dive in. At the top level, our metaphoric equivalent specification might be: "Build us a single-family house." The embedded software version would sound like: "Build us the applications software for a primary flight display for general-aviation aircraft."

Clearly, both top-level descriptions are too unspecific for any meaningful project planning. We need more detail. In the case of the house, the details go down just a few levels. There is general agreement on the approximate level of specification required before implementation can proceed. You typically won't find a blueprint with individual nails, screws, and other hardware described, but one level of abstraction above that is common. The specification stops (and implementation can start) at a level of abstraction where there are conventions established. In the case of construction, those conventions are provided by things like building codes that specify the defaults for implementation details. It is not necessary to specify beyond that level of detail.

In software, however, the level of specification detail would ideally stop at the point at which a compiler could take over, infer the meaning of what we're trying to do, and continue to complete the detailed implementation. Any specification we write at a higher level of abstraction than that is really just software we can't yet compile.

That's a nice theoretical argument, but what does it mean in real-world practice? Software is pure complexity. Once you tame that complexity to a degree that you could write a meaningful specification, your work is almost done.

I once managed a software development team (8 engineers) developing a complex, interactive application. We were required by company policy to

generate a detailed specification for the user interface before starting implementation (a very, very bad idea). After about three months of full-time specification development, we had a 1,600 page document with every possible (we naively believed) user interaction, button, widget, menu, dialog, stroke, command, and shortcut all clearly and accurately specified. We completed the mandated specification review process (an additional three weeks of work), dutifully updated the spec with all the feedback from the corporate review team, and prepared to begin implementation.

After one week of coding work, our specification was practically useless. Even the most primitive of interactive mockups uncovered massive holes and inconsistencies in our specification that had not been detected by weeks of management, team, and peer review. Operations that looked simple on paper were confusing or almost impossible for a real user working with a prototype. Important options were missing from commands and dialogs. Many modes were inconsistent, and some were completely inaccessible. Of course, our specification was accompanied by a complete construction-style project plan with PERT charts, Gantt charts, a work breakdown structure (WBS), resource requirements and a detailed schedule. All of those were now useless as well.

Dwight D. Eisenhower once said "Plans are nothing; planning is everything." At this point in the project, our development team had proven Eisenhower at least half right.

What followed was a classic case study in adaptive management. We wasted a few weeks with futile attempts to keep our ballast of a specification synchronized with our rapidly evolving executable UI prototype. We found that our development schedule time was rapidly evaporating, and that far more effort was being expended keeping our specification up-to-date than developing a workable UI for our product.

Our team then settled on a strategy of software development subterfuge. We "froze" a version of the specification (for management placation) and focused our energy on developing an effective, working UI through iterative refinement of our software prototype. Immediately progress picked up by a factor of five. We quickly converged on a beautiful and elegant interface that was intuitive, fast, complete, and consistent. Remarkably, through an heroic effort by the team, we completed the project in almost exactly the allotted total time on the original schedule. What we produced was almost 100% different from our original specification and absolutely none of the tasks on our original project plan were ever started, tracked, or completed. In a private post-mortem held by the team after code freeze, we agreed that the paper

specification phase had added no value to the process and had probably quadrupled the cost and length of the project.

In our case, the effort to develop the software was far less than the effort to develop a detailed specification. Furthermore, the software itself provided a spectacularly more effective vehicle for evaluating and correcting our design. In the "measure twice, cut once" world of construction, it pays to plan carefully before you pour concrete. In software, however, we have no concrete to pour. You see, software is not a house.

What if you're working on a project that is more straightforward? Perhaps you can accurately break down the development schedule into manageable tasks ahead of time. Can't you then use traditional construction-project planning and management techniques to estimate and track your team's time? If you were building a house, the answer would be yes, but the problem is...

Choose a software development task at random, then ask your favorite software engineer how long it will take to complete. The task can be literally anything – a parser, a licensing system, a communications protocol handler, a new GUI - it really doesn't matter. (By the way, this is a cool trick, and you should try it yourself.) The answer you will get back, no matter what task you picked, will be "about three to four weeks." OK, you think that was a fluke. Pick a different software engineer, a completely different task, and try again. Now are you convinced?

The reason this trick works is still a mystery to most software-scheduling scientists. It is widely believed, however, that it is a corollary of the secret 90/10 rule of software development. The secret 90/10 rule says that software under development will appear to be 90% complete when it is, in fact, only about 10% done. Software engineers don't know the 90/10 rule. If they start on a project, sometime in the first two weeks it will be 10% (actual) complete. If the real development time is anywhere from one to twenty weeks, of course, this will be true. According to the 90/10 rule, though, the system will now appear to be 90% complete. You'll probably even get a "demo".

Also at that point, in the software engineer's mind, the project is "almost done" with "just a week or two of cleanup and bug-fixing." (Ergo – "three to four weeks" for the original estimate seems just about right.) For the next ten or twenty weeks, however, the scenario will be increasingly grim. Each week, the project will be "almost done," with "just a few more loose ends to clean up." Finally, on the project manager's last week with the company, the

decision is often made to ship the system anyway and start scheduling the first patch release.

In construction, the workers are completing tasks exactly like those they have done many times before. Apparent progress and actual progress track almost linearly. When the framing of a house looks half done, it is half done. In software, developers are almost always building something they have never built before. (If they had built it before, it would already be done.) Combine that with the secret 90/10 rule, and we've got a monumental problem for managers trying to schedule and track software development tasks.

Even if we could create an accurate "to do" list for our software projects, and even if we could get accurate estimates for individual tasks, we're still not out of the woods. There is one last important way in which our project planning metaphor melts down. In construction, most tasks have end-start dependencies. If task "B" has a dependency on task "A", task "A" must be completed before task "B" can begin. The foundation of a house must be completed before we can begin framing. Framing must be completed before we can wire. Wiring must be done before we can install wallboard. These dependencies are the fundamental reason for creating a PERT chart. It derives schedule concurrency and critical paths (and ultimately calculates schedule length) based primarily on these dependencies and estimated task durations.

In software, however, most dependencies turn out to be vague, middle-middle dependencies. Most development tasks can actually be started in parallel, but almost none can be completed without some degree of progress on others. This makes a PERT chart an interesting ornament for your team's coffee lounge, and an excellent PowerPoint slide to show to management, but absolutely useless for schedule planning and tracking. Although a PERT may work great for your homebuilding contractor, software is, well, different.

If you work for a pure software company, you may have long ago conquered these issues. In the embedded space, however, software development is often managed in parallel with hardware development, using a project planning infrastructure that is hardware-biased and overseen by management with little software development insight. The mandating of metaphor-based methods like those listed above can cause disastrous effects in software development, and ultimately in product delivery and success.

In many of these embedded systems companies and projects, these effects are seen every day. Different teams deal with them differently, however. In some, faux project plans are carefully maintained and presented to

management while actual development proceeds quietly at its own pace and in its own way behind the project manager's well-guarded firewall. In others, constant combat erupts between developers and managers as communication breaks down between developers trying to follow incongruous instructions and managers trying to fit fluid projects to static standards.

So, if software isn't a house, then what is it? Are there sane and sensible methods that can be applied to solve the software scheduling dilemma? In part two of this series, we'll take a look at some of the proven ways we can produce a practical embedded software project plan, track our progress in a meaningful manner, and generate useful organizational learning that can help us predict future projects even more accurately.

*Kevin Morris, Embedded Technology Journal*

*November 1, 2005*