Software Development and Linearity
(Or, why some project management methodologies don't work)
Part 2

R. Max Wideman
AEW Services, Vancouver, BC, Canada

This paper was first published by ICFAI PRESS, Hyderabad,
in Projects & Profits Special Issue, March 2003, p18

Introduction

In Part 1 of this paper we described the operating environment of information system (IS) departments working in the public and private sectors. In particular, their need to maintain, enhance or upgrade existing software systems, or respond with entirely new software to satisfy new corporate business initiatives. We suggested that their operating environment generally includes a variety of stakeholders with differing goals, a lack of stable requirements, and is often subject to ponderous management direction.

Further, their software development programming itself is not necessarily done "in-house", but rather, these services may be acquired from outside suppliers under contract. This adds difficulty because the corporate acquisition process may well be that of traditional capital project acquisition based on a fixed price quotation for a total product, even though the details of that product are often highly fluid.

In spite of the cry that "Software development projects are different!" we questioned that assumption. In our view, the real issue is not that software projects are really different but that the nature of the product requires an appropriate project management methodology. The crux of the matter is the project's life span, which we described for reference purposes, and how the work should be properly advanced and controlled. In this light, we examined a number of methodologies such as the Project Management Institutes so called "PMBoK", the "waterfall", the systems engineering approach, and the "spiral" model, and listed the strengths and weaknesses of each.

In this Part 2, we will also examine "Rapid Prototyping" and then move on to suggest reasons why "linearity" really doesn't work for IS department software development. We will also look at some of the "people" problems associated with software development and try to suggest some answers – considering that the answers have probably been around for a long time, if only we would look at them.

So, what is the best approach, or methodology? Well, it all depends – see our conclusions!

Rapid Prototyping

Rapid or "evolutionary" prototyping is another effort to overcome the difficulties of the linear approach. Essentially it says, forget all that up-front documentation and, working with the users, get right into the development. It is also known as "Joint Application Development". The idea is to talk to the customer, build a prototype, display it to the customer, get their reaction and modify accordingly. Repeat the process, correcting errors and gradually adding functionality, until the customer is happy with the

product. The underlying concept is that it is more time and cost effective to work with the product than to spend it getting the requirements documentation exactly right.

Good features

- The cost of intermediate deliverables is eliminated because time is spent on creating product rather than a lot of paper.
- Software development documents are difficult for users to interpret anyway. This is unlike architectural building design and working drawings, from which most people in the construction business can get an idea of what the constructed work will look like.
- Right away, you get the programming team working — and enthusiastic.
- You bypass the business of establishing accurate project assumptions.
- Direct user involvement leads to a high level of "buy-in", even to the extent of overlooking some things they may not like.
- Thus, user "requirements" evolve as the project progresses.

Not so good features

- No one knows how many build-and-display cycles will be required.
- Hence the approach is difficult to plan and establish a schedule and budget for executive control.
- The approach breaks down when large complex systems are involved where many development teams may be involved in the entire system.
- It may also break down where many different stakeholders are involved.
- In the end, all you may end up with is still a prototype.
- It is difficult to bring the project to a close because there is always the temptation "To do just one more cycle."

This looks like a good practical technological approach and works well provided that money for the work is plentiful. However, from a project management perspective, the customer may never be satisfied, and the project just never ends because the project manager has no leverage to make it so. The short-comings described all represent serious project management schedule and cost risks.

Reasons why linearity doesn't work for IS-department-type software development

Software project content is much more capricious. In construction work the content, once established, is relatively straight forward and less likely to change in any major way such as the height of a building or length of a road. Relatively small changes can be handled by well-established change procedures. For most construction, especially fixed-price contract work, the amount of work involved is clearly understood. Software differs for several reasons.

- Software is interactive and has a lot of "behavior", i.e. functionality, which makes it harder to specify. For exactly this reason, users, the ultimate customers, find it difficult if not impossible to specify their exact requirements at the outset. "Give me something to look at and I'll tell you if I like it" they cry!
- Consequently, it is not possible to establish cast-iron specifications covering subsequent execution work.

Scope is a more active variable. In the traditional construction tetrad tradeoff of scope-quality-time-cost, scope is usually relatively fixed. You would not, for example, contemplate building half a bridge in order to provide twice the number of lanes over that half. It just would not make sense. In software such a compromise is more likely, and plausible, especially if unexpected risks are encountered.

It is almost impossible to "freeze" software specifications. Linearity depends on freezing work done before moving on to the next phase, but in reality nothing stays really frozen. As we've seen, the customer is rarely able to completely specify requirements sufficient for development. Similarly, the designers are unable to tell whether the various technologies to be used in the final solution will in fact work together. The complexity of software development, and the pace of change of software tools, makes it almost certain that there will need to be fixes for flaws in assembly.

Besides, when the users get their hands on the software, they will inevitably discover better ways to use it or have it work better. User interaction with the software is a very personal thing. A successful interface may well depend on what other software is familiar to the user. If it doesn't work the way expected, i.e. it is "counter intuitive", changes will be necessary such that users finally "own" the result and call it "user-friendly"..

Some scope creep is inevitable. According to Hal Helms: "Scope creep is the pejorative name we give to the natural process by which clients discover what they really want. This puts our attempts at "requirements gathering" in a different light. Most project managers try their best to discover what clients want at the beginning of the project. They use meetings, questionnaires, personal interviews — and still, the most common experience for developers delivering a final product is customer dissatisfaction. It may come as a slap in the face: "This is no good"; or it may be couched in gentler terms: "You know what would be nice?" but the same message is being delivered — we aren't giving clients what they want."[1]

Building software is all intellectual work. The results are "abstract" in the sense that the real value of the product is in its performance not in its tangible evidence, i.e. the hardware it sits on. It is the result of brain work, not brawn work, the outturn of a different kind of worker who needs to be managed differently. At the same time, it is more difficult to establish how many lines of code will be required to achieve the required functionality, given all the possible permutations and combinations. Consequently, it is more difficult to assess interim progress or, indeed, when the final product is actually complete.

Software scheduling logic is not self-evident. In construction, you cannot put the roof on until you have built the walls. Software development is more like road building – you can start anywhere you like, but smart contractors start with building the bridges, the most difficult and possibly risky parts, but in a logic sequence that is not linear.

You cannot schedule software projects by effort calculations. You might be able to halve the time for constructing a brick building by doubling the number of brick layers, each working their own section until they meet at the junctions. However, you cannot do the same with software because of the interaction necessary. As you increase the number of programmers on the same element, the number of interactions increases exponentially until interactions exceed the rate of progress and the work collapses. To quote an old saying: "You cannot reduce a pregnancy to one month by impregnating nine women."

The exposing of risk tends to be different. In construction, the biggest risks tend to be associated with site conditions which can be dealt with early on. Subsequent risks having to do with workmanship are mostly visible as the work progresses. True, "latent" risks, discovered after completion, are a possibility but tend to be relatively rare. In the case of software development, risk sources, e.g. unanticipated interactions that don't work, are more likely not revealed until later. That is, when the whole system has been in use for some time and users become sufficiently familiar to push the application to its limits.

People problems

Scott Ambler, promoter of the Agile Data Method, provides some interesting insight into the software developer's working environment.[2]

Different visions and priorities. Developers focus on the specific needs of a single project and strive to work in isolation from the rest of the organization. Administrators focus on the overall needs of the enterprise, sometimes to the detriment of individual project teams, yet the scope priorities and issues of each are all different. To make matters worse, project stakeholders, including users all the way up to senior management, have varying priorities and visions as well.

Over specialization of roles. Specialists tend to be too narrowly focused. Because they are specialists, they may have difficulty relating to other professionals.

Process impedance mismatch. A number of recent approaches to software development advocate an incremental and iterative approach. However, many IS/IT managers still view the process as a serial or near-serial process. Consequently, there is a process mismatch necessitating a cultural and organizational change within the organization.

Technology impedance mismatch. Developers work with objects and components whereas data professionals work with databases and files. Software engineering principles form the underlying foundational paradigm for objects and components whereas set theory forms the underlying foundational paradigm for relational databases (by far the most popular database technology). Because the underlying paradigms are different the technologies don't work together as well as they should.

Ossified management. The technology and techniques used by IT professionals change rapidly. As managers progress up the corporate hierarchy they deal less with technology and more with people issues. It doesn't take long for their previous technical experience, on which they base decisions, to become obsolete.

Organizational challenges. Poor communications and "office politics" hurt development efforts just as badly as they hurt other enterprise efforts.

Inappropriate documentation. Either too much, or too little, documentation may prevail depending on management will. Too little and it becomes impossible to retrace earlier steps; too much and no one reads it.

Ineffective software architecture efforts. Software architecture is not a well established discipline, let alone a profession as in building construction. A lot of people simply don't know where to start. And, if they do, the resulting models are soon abandoned.

Ineffective development guidelines. Some organizations struggle to develop guidelines for all their IT people to work to. But, as with any guidelines including project management guidelines, some people are not aware of them; if they are they don't understand them; or if they do they consider them obsolete; or they are otherwise unwilling to follow them anyway.

Ineffective modeling efforts. Some people focus on a specific aspect to produce models that look very good only to discover that they do not reflect the real world, especially if past experience and practices are not factored into account.

So what is the answer?

In response to the difficulties and failures described, to say nothing of the frustration felt by software development team members, a number of technological approaches have emerged or are emerging. For instance: Agile software development, Extreme programming, and so on. As an example, the manifesto of the Agile Alliance reads as follows:[3]

Quote
We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

- Individuals and interactions over processes and tools
- Working software over comprehensive documentation
- Customer collaboration over contract negotiation
- Responding to change over following a plan

That is, while there is value in the items on the right, we value the items on the left more.
End quote

A very laudable vision. However, which ever way you look at it, project management is a process, and one wonders whether such a manifesto is more designed to assuage programmers discomfort with cost and schedule constraints than to solve the problem of executive project control. As we described it at the beginning, a project has a "start" and a "finish" and a "bit in the middle" and that bit-in-the-middle consists of planning before doing. That all adds up to serialized activities, and this should not be surprising because "process" is defined as "A set of partially ordered steps intended to reach a goal."[4]

I don't believe that any clear answers have yet emerged, but there do appear to be some pointers:

- Software development is a problem-solving exercise.
- Schedule adjustment is not amenable to effort loading because there is only a narrow range of "right number of people for the job." Too few and the project will be extended by under-staffing. Too many and the project will be bogged down by excessive communication.
- The inherent uncertainty in software development is closer to the high end of technological risk, such as research and development, than it is to traditional building construction. Cost and schedule control and consequent contingencies must reflect this reality.

- Project management does require some serialization to reach a required destination.
- Therefore, good solutions must be arrived at by iteration, not just for the product but for the planning for the product as well.

The idea of iteration, prototyping, or trial product recycling should not faze any project manager of experience. The technique is well know in the construction industry. Some twenty years ago I worked on a very innovative light rapid transit system for Vancouver, Canada. The concept called for driverless, fully automated computer-controlled light aluminum train cars powered by linear induction motors on 21 km of elevated track. The majority of that track would be constructed of pre-tensioned concrete track beams, pre-cast to a standard that would eliminate "cusping" of the track at the bridge-beam junctions.

To enable a smooth high-speed ride the whole system in this hilly terrain was designed to a continuous horizontal and vertical curve generated on a computer program written for the purpose. All these innovations were areas of high technological risk. To test the concepts, a full-blown trial section, i.e. a prototype, of one kilometer was constructed at the outset. This test section resulted in a number of changes, such that the final project was commissioned ahead of time and survived flawlessly its first full-scale, public free-ride overload test program. The system has since operated with remarkably little down time.

Another example of prototyping is more common. It is standard practice in the apartment and hotel building development business to construct a typical show suite or hotel room even before the structure is completed. This is to make sure that furniture will fit, circulation areas work and even furnishing, fittings and colors blend and are acceptable to knowledgeable marketing people in the industry.

But perhaps the final pointer in software projects is the most telling:

- It is typically necessary to enter the **next** stage of software development to establish the consequential effects of the **previous** stage before it is possible to validate the acceptability of that previous stage. This may well result in going back to that previous stage and modifying it.

This is **a significant departure** from the linear mantra governing construction projects.

Conclusion

From the forgoing examination, we conclude that software development projects, in the environment we described, really are different from "traditional" project management, i.e. projects involving well-established technology. So, if you are an IS software development project manager and your first need is to identify the best methodology for your project, here are some suggested questions for determining the best selection.

- Who is the sponsoring organization, how much documentation do they usually expect and are they willing to support it? This will establish the degree of rigor to be applied to the whole process and the extent of data management required.

- How big is the project, specifically is it "complex", i.e. how many elements are there in the system and/or how many stakeholders are there that need to be satisfied, and in what way? This will determine the number of interfaces and communication channels to be coordinated and

hence the level of project management required.

- What is the risk "profile" of the project, that is, how "risky" are the various elements of the project's scope, or how risky are any of the proposed technologies that will be involved? That is to say, does the customer really understand what they want, or only have a general objective? This will determine the priorities for tackling the system elements to mitigate the risks as early as possible. It will also indicate the level of risk management and contingency allowances required.

- Which of the four project management constraints of scope, quality, time or cost predominates? That is, which of them has real priority for the top position and which, in the last analysis, can give ground? For example, is there a "time window" such as meeting a market opportunity? This will determine the pace at which the project must be conducted and consequential constraint on the extent and number of iterations permissible.

- Another way of asking the last question is: What are the measurable key success indicators (KSIs) by which the product will ultimately be judged and against which decisions can be taken during the project? And, what is the order of priority amongst the KSIs? This will establish the basis for effective project management and necessary executive control.

Given the answers to these questions, a suitable methodology may suggest itself. In any case, your solution to the selection dilemma should include some judicious compromise between prototyping or iteration, and serialization. If you can surmount that hurdle then it only remains to apply standard project management techniques. That is, adopt a suitable organization, leadership style and communication, and apply the right competent skill sets to the selected methodology. You will then be well on our way to a successful project.

Meantime, instead of lauding one software development methodology over another, perhaps we should be examining each of them in the context of the variables identified, to establish the best approach to methodology selection. That is, how to select a software development methodology for a given project environment that leads to the highest probability of a successful outcome. On a large project that could well include several of the methodologies described above, but applied separately to different elements and/or to progressive phases of the project.

Max Wideman

---

[1] Hal Helms, http://www.alistapart.com/stories/scopecreep/ Issue #150, September, 2002
[2] Scott Ambler, abstracted from http://www.agiledata.org/essays/vision.html#WorkingTogetherIsHard, 2002
[3] http://agilemanifesto.org/
[4] Rational Unified Process Glossary, 2000.