Software Development and Linearity
(Or, why some project management methodologies don't work)
Part 1

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

The focus of this paper is on software development that flows from the corporate requirements of information system (IS) departments working in the public and private sectors. IS departments are typically created to serve organizations that range in size from medium to large. Unlike other areas of software development, such as "shrink-wrap" or "commercial-off-the-shelf" software development, the needs of IS departments are to maintain, enhance or upgrade existing software systems, or respond with entirely new software to satisfy new corporate business initiatives.

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. However, the software development programming itself is not necessarily done "in-house". Instead, these services may be acquired from outside suppliers under contract. This makes it even worse 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 not yet entirely known. Thus, responding software developers face even bigger hurdles.

Understandably, these service-provider developers cry: "Software development projects are different! Software development just doesn't work the same way as "regular" projects!" Well are they? After all, for all projects time is linear and there is no "time undo" button to undo anything not to your liking. The best you can do is go back and fix it.

And projects are essentially linear, too. That is, they have a start and a finish "and a bit in the middle" as Professor Rodney Turner[1] is fond of saying. So, what's the problem? The problem is that in spite of the best efforts using some of the best methodologies, project software development results are still disappointing in terms of cost and schedule. So, let's examine some of the better-known methodologies in the context of software development-type project management.

However, before doing so, we must first be clear on what constitutes a project result that is "not disappointing". For this we turn to the definition of "A well-managed project". "A well-managed project is one that is optimized for effectiveness in its planning phases but emphasizes efficiency in its implementation phases, that include commissioning, startup and close out. [D02129]".[2] To this, in the case of software, we should add something like "And, in the eyes of the customer, is perceived as satisfactory in use."

This definition has important implications for software project management. It is about planning and

control. That is, in a project life span that has four major "generic" phases, in the first phase of "Conception" you establish the "vision" for the product. You also justify the project in a "Business Case" document that requires executive or upper management approval before proceeding further. In the second phase of "Definition" you determine content based on requirements, and develop a plan for execution that you describe in a "Project Charter", project brief or similar document.

This, too requires executive approval, especially since this approval will set aside major funding for the work of the project. In the third phase of "Execution", that is the actual software program development, you create the product, and test and verified it for "customer satisfaction". In the final phase, you transfer the product to the users, complete with documentation and any training needed, and you formally close the project, following acceptance and approval by the executive. We call this aspect of project management "Executive Control".

Along the way, you will inevitably encounter the need for decisions, compromises, trade-offs and changes. To support sensible decisions, you should know the order of priority of the four core project management variables of scope-quality-time-cost and the level of risk associated with each. For example, in an air traffic control system, quality would obviously have the highest priority at the expense of time and cost. On the other hand, software designed to facilitate some public exhibition must meet the opening date so time would obviously be at the top of the list. An accounting or tracking system project, whose scope includes a number of features of varying degrees of desirability, could be scaled back if limiting cost against budget is the major concern, and so on.

In selecting a software development project management methodology, there is also the issue of complexity. By complexity we mean the number of separate stakeholders you need to satisfy and/or the number of system elements that you have to integrate. It is against this background that we examine some of the more popular project management methodologies.

The Project Management Institute's Guide to the Project Management Body of Knowledge

We start with this one because it is, perhaps, the most widely known. However, it is more of a cult than a rigorous methodology. As the Guide itself says "The primary purpose of this document is to identify and describe that subset of the PMBOK® that is generally accepted. Generally accepted means that the knowledge and practices described are applicable to most projects most of the time, and that there is widespread consensus about their value and usefulness. *Generally accepted* does not mean that the knowledge and practices described are or should be applied uniformly on all projects; the project management team is always responsible for determining what is appropriate for any given project."[3]

Notwithstanding, the Guide has all the hallmarks of a methodology, and in the absence of anything else is often used as such. The frequent basis of this methodology is the infamous set of process groups: Initiating; Planning; Executing; Controlling; and Closing, which are intended to apply to each and every phase of the project life span as a management cycle.[4] Understandably, with these labels, this series is often taken as the set of phases of the project life cycle itself. The consequence is that "Controlling" looks like a "phase" to be entered into and then abandoned when done.

In reality, "control" should be exercised throughout the project life span, but the Guide does not discuss

overall project control in its discussion of project life cycles. As Forsberg et al have clearly illustrated,[5] a distinction must be drawn between "situationally" applied management elements and the "sequential" project cycle. The control process and its supporting components of baseline planning, measure, compare and correct, are clearly situational and hence cannot, in any way, be considered as a "phase".

Beyond this, and other than a general discussion of several displays of project life cycle, the expectation of each "knowledge area" discussed in the Guide is that you first figure out the project's requirements (in the knowledge area) and then move on to planning and then execution. Well and good, but this is a linear progression that does not seem to serve the software development environment.

The "Waterfall" process

The waterfall process is essentially a linear process as shown in Figure 1.

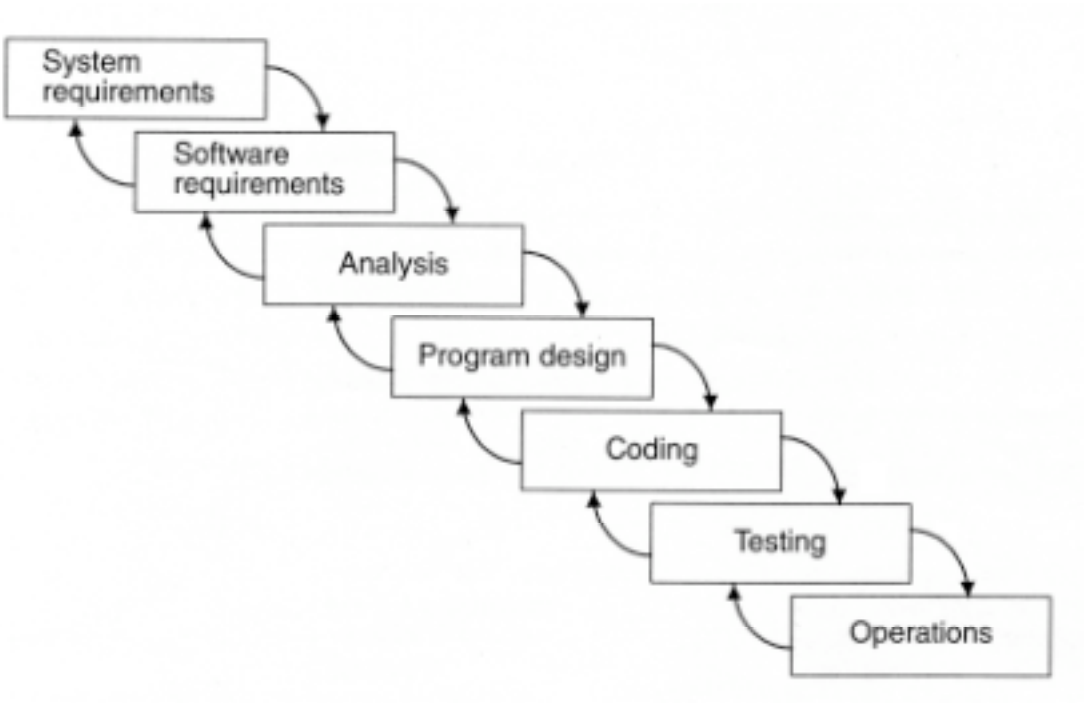

Figure 1: The Waterfall Life Span (Royce 1998)

To explain this phenomenon we can hardly do better than to quote Murray Cantor extensively.[6]

Quote

Software development may seem to have a lot in common with construction or engineering projects, such as building a bridge. A set of activities must be carried out in what seems to be a natural order: requirements gathering, design, building, testing, and finally putting the bridge into general service. From then on, the bridge goes into operation and maintenance. For software, the usual activities are recast as requirements analysis, architecture, design, code and unit test, integration, and system test.

Tasks are serialized in construction projects. For example, you cannot lay the foundation for a building until the excavation is completed and approved. The flooring task must not begin until the joists are

inspected and signed off. Early approaches to software development tried to follow the same discipline.

1. The analysis team captured and documented the requirements.
2. When the requirements were approved, the design started.
3. When the design was approved, coding began.
4. Each line of code was inspected. If it was approved, it was allowed to be integrated into the product.

This is the dreaded waterfall process. It was once touted as a way to make software development cost-effective. The thought was that if coding began before the design was approved, some coding effort would probably be wasted. In practice, **. . .** the construction mentality inherent in the waterfall process has led to some spectacular software development failures.

. . .

The first problem with this approach is that the software development tasks are not as easily planned and assessed as those of a construction project. It is simple to know what it means to be half done with painting a bridge. It is difficult to know when you are half done with writing code. The amount of time it takes to paint each part of a bridge is easy to estimate, but no one really knows how large the final code will be, and no one knows exactly how long it will take to write and debug any particular piece of code.

A second problem is caused by the serialization of the activities: completing one activity before starting the next. To complete an activity implies that the outcome is perfect and that the staff assigned to the activity can move on to the next project [but] you cannot be sure whether, say, the requirements document is finished. In fact, experience has shown that you can be sure it is not finished. Over the life of the project, shortfalls in the requirements document will be discovered [or in the requirements or design specifications.] **. . .** Each time the documents are reviewed, new problems arise, doubts are raised, gaps discovered, and questions are asked that cannot be answered. Management is disciplined, insisting on quality and holding to standards. They want flawless documents that describe inhumanly complex systems. Their expectations cannot be met and, in the end, a lot of money is spent with no useful results. There is plenty of blame to go around.

End quote

Good features

- The waterfall approach has been around for a long time, and many people are familiar and comfortable with it.
- It is simple and easily understood.
- It does recognize the need to move one stage at a time and recycle back to the previous stage to validate the stage outcome

Not so good features

- The waterfall approach does not satisfy the requirement for executive control that we described in our introduction.
- It is very difficult to manage under conditions of complexity.
- In the waterfall approach, integration and testing is generally left until the end. That's when "all the chickens come home to roost", with disastrous effect on project schedule and cost.

As Shelley Doll puts it:[7]

- "Another potential danger is that you won't know if the solution is successful until very close to launch, leaving little time and room for correction. Oversights and flawed design work can seriously affect your launch date.
- "Other criticisms of this model include the fact that there's no room for feedback anywhere in the process, except at the end of a phase.
- "Also, there's no room for changes once development has begun.
- "Finally, if system testing shows that capacity or performance isn't up to snuff, it may be impossible to correct."

In short, the waterfall approach may be fine for smaller projects but not for projects of any size requiring effective executive control.

The Systems Engineering approach

The systems engineering approach is also a linear process. Perhaps the best example is the "Vee Model" as depicted in Figure 2 and described in detail by Forsberg, Mooz and Cotterham.[8]

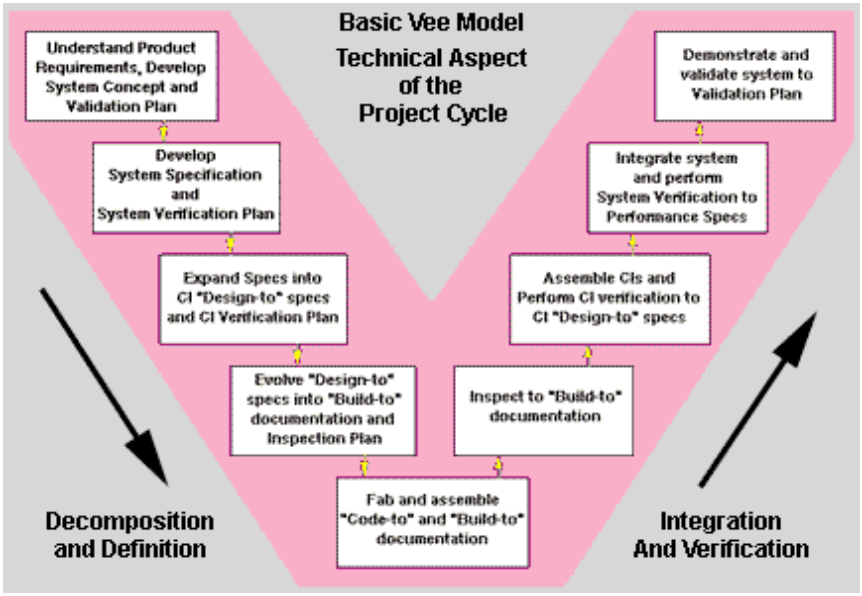

Figure 2: The "Vee" Model (Mooz, et al, 1996)

But as Cantor explains:[9]

Quote

Systems engineering is concerned with designing, specifying, and verifying the implementation of components of complex systems. It was developed to meet the challenge of building large aerospace systems such as the Space shuttle and NATO's command and control systems. These systems are often so large that their development must be distributed to several companies. One firm, the prime contractor, is assigned overall design and integration of the system. The prime contractor's systems engineers gather and analyze the system requirements, then design the system by decomposing it into a set of subsystems.

. . .

The subsystems may be developed by subcontractors, who are obligated by contract to meet the requirements and specifications for the subsystems. The developed subsystems are delivered to the prime contractor for integration into the total system. The prime contractor's systems engineers verify that the subsystems meet their requirements and oversee integration.

. . .

Initially, the systems engineering approach may seem to be a reasonable approach to developing large software systems. Many systems engineering activities apply to software development:

- Requirements gathering and analysis
- Architecture (identification and specification of the systems)
- Subsystem development
- Integration and verification

However, the systems engineering approach is based on assumptions that do not apply very well to software development. Systems engineering assumes the requirements are stable and the interfaces can be specified adequately before implementation. In practice, the systems approach is a particularly rigid variant of the waterfall approach and its lifecycle, and it inherits all the weaknesses of that approach:

- Classical systems engineering is document driven. Systems engineering efforts are driven by formal documents that are developed by engineers and implemented by developers. The problem is that the documents are never correct. The systems engineers rely on their ability to make the document correct. They focus on creating specifications that, once approved, are adequate for determining the design of each subsystem. Any change to the document is managed by a cumbersome change control-board system.

End Quote

Good features

- Most people who have an engineering background are very comfortable with the systems approach
- It is very good wherever it is possible to describe, i.e. specify, the requirements with a high degree of certainty
- The acquiring authority requires a thoroughly well-documented track record or audit trail
- Consequently, it is popular with big government departments,
- Where money is not the limiting criteria, though competitive bidding might be.

Not so good features

- The process is heavy on documentation
- It assumes that it is possible to arrive at near-perfect documentation that is complete, and is truly representative of the ultimate "requirements"
- And can be frozen, and the authors, i.e. the stakeholders, can be held accountable to those requirement specifications.

In summary, this approach does not fit well with the IS department's environment, sometimes described as "the voyage of discovery", that we described in our introduction.

The Spiral Model approach

It seems that "spiral development" means different things to different people. The concept was first described by Barry Boehm[10] as an iterative waterfall in which each iteration provides increasing software capability. This concept is shown in Figure 3.

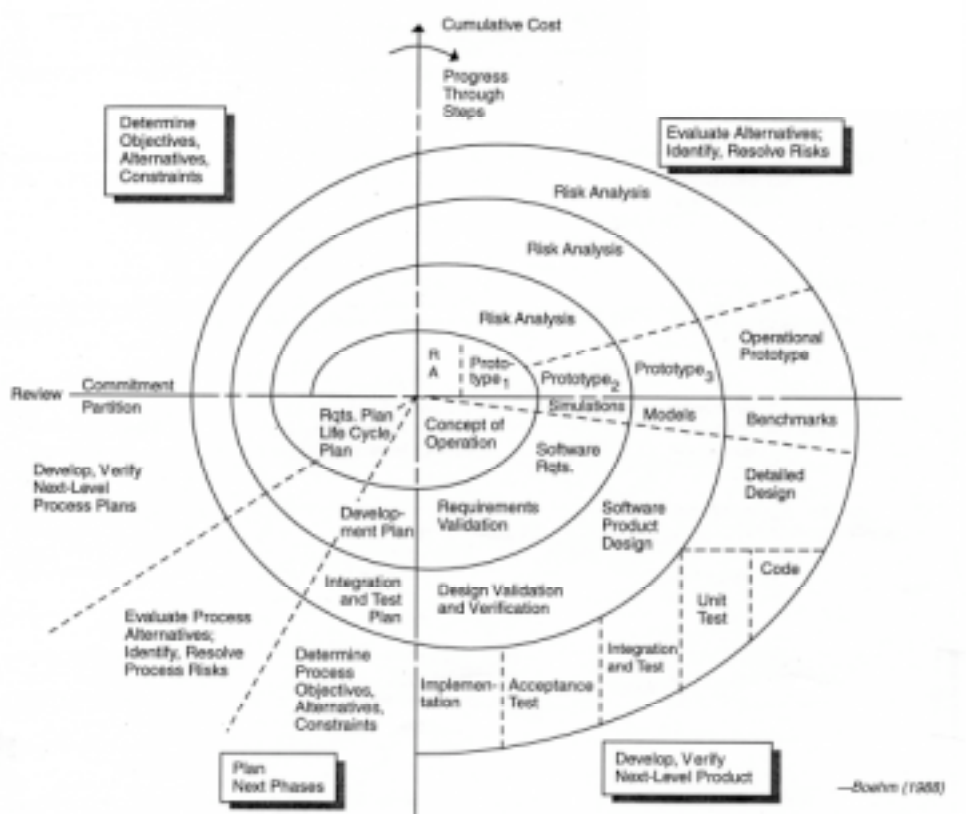

Figure 3: Spiral Development (Boehm 1988)

However, today others may use the term to mean "Rapid Prototyping", or even "Anything but waterfall". For the record, a "spiral" model of software development is described by Muench.[11] It consists of a spiral divided into four quadrants. Each quadrant represents a management process, namely: Identify; Design; Construct; and Evaluate.[12] The system goes through four cycles of these four processes:

1. Proof-of-concept cycle – define the business goals, capture the requirements, develop a conceptual design, construct a "proof-of-concept", establish test plans, conduct a risk analysis. Share results with user.
2. First-build cycle – derive system requirements, develop logic design, construct first build, evaluate results. Share results with user.
3. Second-build cycle – derive subsystem requirements, produce physical design, construct second build, evaluate results. Share results with user.
4. Final-build cycle – derive unit requirements, produce final design, construct final build, test all levels. Seek user acceptance.

Good features

- In this approach, the entire application is built working with the user.

- Any gaps in requirements are identified as work progresses into more detail.
- The process is continued until the code is finally accepted.
- The diagrammatic representation, i.e. the spiral, does convey very clearly the cyclic nature of the process.
- And it also conveys the progression through the project life span.

Not so good features

- This approach requires serious discipline on the part of the users.
- If the users are not responsible for the schedule and budget, as very often they are not, executive control can be difficult.
- For a software developer working under a firm-price contract, it may be impossible.
- The model depicts four cycles. However, if cycles are added indefinitely for "just one more tweak" then eventually, as Cantor says, "Everyone gives up in frustration!"
- He might have added "Or, the time and money runs out."

All things are possible, but some are less likely in practice. The problem here is that authority and responsibility are divided making executive control difficult. Under these circumstances the spiral can become a project that just keeps going round in circles.

To be continued

In Part 2 of this paper we will take a look at Rapid Prototyping and then move on to examine 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!

Max Wideman

---

[1] Rodney Turner, author and editor of The International Journal of Project Management, and Professor of Project Management at Erasmus University, Rotterdam

[2] Project Management Guidelines (Private BC Corporation), 1995, Wideman Comparative Glossary of Project Management Terms v3.1, 2002

[3] A Guide to the Project Management Body of Knowledge (PMBOK® Guide), 2000 Edition, Project Management Institute, PA, 2000, p3

[4] Ibid, p30

[5] Kevin Forsberg, Mooz, M. and Cotterham, H., Visualizing Project Management, 2nd Edition, Wiley, 2000, p43

[6] Murray Cantor, Software Leadership: A Guide to Successful Software Development, Addison-Wesley, NY, 2002, Appendix A.2, p153

[7] Shelley Doll, Waterfall for new managers http://builder.com.com/ July 24, 2002

[8] Kevin Forsberg, Mooz, M. and Cotterham, H., Visualizing Project Management, 2nd Edition, Wiley, 2000

[9] Murray Cantor, Software Leadership: A Guide to Successful Software Development, Addison-Wesley, NY, 2002, Appendix A.2, p162

[10] Barry Boehm, A Spiral Model of Software Development and Enhancement, IEEE Computer, 1988

[11] Dean Muench, The Sybase Development Framework, Sybase, CA, 1994

[12] A Guide to the Project Management Body of Knowledge (PMBOK® Guide), 2000 Edition, Project Management Institute, PA, 2000, p17