



April 2005 Issue

Which is the Right Software Process for Your Problem?

by Kal Toth, Portland State University



Abstract: Proponents of lifecycle processes boost the benefits but rarely discuss which problems they best suit. This article presents an approach for selecting processes according to the attributes of the problem at hand.

The Problem

Previous SAO articles have highlighted the characteristics and benefits of several software processes and methodologies and the literature and textbooks are filled with many others to consider. From the traditional Waterfall to newer methodologies such as Agile and XP, proponents describe their properties and outline their benefits. Some methodologies claim that they can be customized to fit large, highly critical applications as well as very simple ones. But rarely is there much said about which sort of problem or application they best address or how to adapt the process to fit the problem at hand.

Can one use the advocated process for any problem? What about trade-offs among alternative process models? Can the process be tailored to address specific needs or system constraints? Can I combine the best features of two process models to produce the software product attributes I am really after?

The prudent software engineering manager needs to filter all this background noise and “marketing” and keep the following in mind:

- choosing the right process will yield positive economies and competitive advantage
- choosing the wrong process will lead to project over-runs, failures, lost business

Because projects undertaken by a company - even within a given division - can rarely be characterized by the same or similar attributes, there are numerous opportunities for making correct and incorrect process choices. Adopting the same process for all projects is unlikely to be a good recipe given the diversity of customers, range of applications, and the rate of technology change going on out there. The company should therefore expect responsible personnel to thoroughly understand the attributes

of each of their software projects and ensure that processes are selected and tailored to the project's unique needs as early as possible.

So how does the software engineering manager go about selecting the one that is most suitable to the problem at hand and the company's capabilities? In this article I provide consolidated insight into such process selection.

Process Selection and Tailoring Criteria

I see three dimensions to establishing an effective process for a software project:

- **Infrastructure:** This is a fairly loaded term to address prerequisite technical, human and environmental factors that critically affect software project success and failure. It covers the acquiring of leadership and management skills, jelling and leading teams, collaborating with customers and users, and deploying cost-effective development tools and technologies.
- **Process Lifecycle:** This is the macro-process or model that shapes the general approach and strategy the software team will follow to carry out the project. Key process models include, but are not limited to, the waterfall, incremental, iterative (a.k.a. spiral), evolutionary and Agile (e.g. XP) process models.
- **Key Process Areas:** Often referred to as "practices", these are lower-level processes and techniques that guide day-to-day activities and tasks conducted by members of the software development team. They include project management, configuration management, quality assurance, requirements engineering, architectural design, detailed design, testing, reviews and inspections, risk assessment, formal methods, and so on.

My focus in this article is "process lifecycle" selection. In a future article I plan to discuss and provide guidance for tailoring key process areas such as those mentioned above.

Process Lifecycle Criteria

Several factors characterizing your problem or application can influence the selection of the most suitable lifecycle process. I have chosen a short-list of key influencing factors, "drivers", that I believe are most relevant for examining trade-offs among lifecycle processes.

Maintainability: Baselined requirements, stable designs, consistent coding standards, baselined regression tests, and adequate documentation contribute to software maintainability. Highly critical and complex systems often have demanding maintainability requirements as they are very concerned about "total lifecycle costs" – both during development and during operation. On the other hand, adhoc programs for

experimental and concept validation purposes care little about down-stream effort and cost effects.

Application Domain: High-end applications include safety, reliability, security and availability and other critical systems; moderate applications include business and enterprise applications; low-end (low criticality apps) include beta and field trial services, proof-of-concept prototypes, and vanilla-flavored web-sites.

Size / Complexity: For simplicity, let us consider that software size and complexity are relatively linear with respect to each other. Let's assume that large, complex software projects exceed 500K LOC, small, simple projects are less than 10K LOC, and medium-sized projects range in between.

Requirements Uncertainty: This too is difficult to quantify. But let us assume that requirements are highly uncertain if they are only verbalized or if they have been expressed in a few pages with little input from users. Meanwhile, highly certain requirements have involved some up front studies, user inputs, possibly have been prototyped, and have been scrutinized by third parties. Again, moderately certain requirements are somewhere between.

Progress Visibility: Visibility into progress can be achieved through demonstrations and through documentation – both have their shortcomings. Demonstrations are favored by many customers but if not supported by other measures of progress can give the illusion of more progress than is actually being achieved. Documentation is harder to translate into real progress but simplifies contracting – hence it is favored by those with accounting and legal mindsets. For this discussion we will assume that ongoing and frequent demonstrations to customers and users accompanied by reasonable levels of documentation can provide high progress visibility. And infrequent demos plus scant documentation provide little or no visibility into progress.

User Involvement: User involvement can be with respect to developing requirements specifications, validating requirement specifications, inspecting prototypes, supporting detailed design and development, reviewing specifications, and accepting released products. Users heavily committed to supporting three or more of these areas can be regarded as “highly involved”. Marginal support of only one of the areas can be considered “low” involvement.

Requirements Volatility: This factor is also difficult to quantify and requires a good understanding of the problem domain, especially with respect to the maturity of the customer, users, and the development team. If the application is unprecedented or if the problem is in an emerging area, stakeholders will not be sure what will change until parts of the system have been fielded. Often, but not always, requirements uncertainty will go hand-in-hand with requirements volatility (if they are uncertain, they could very well be volatile later on).

Urgency: Demanding market forces will put pressure on a software project to create and release software functionalities and features as early as possible. This will lead to lots of schedule pressure. High urgency, of course, needs to be moderated for mission-critical projects when it comes to human safety and financial transactions where errors cannot be tolerated.

Characterizing Each Lifecycle Process

I have characterized each lifecycle process in terms of the above criteria in the table below. To simplify the analysis, I have chosen to use three-level attributes for each as follows:

H=High; M=Moderate; L=Low ... these terms can be directly interpreted except perhaps for:

- Application Domain where H=safety/reliability-critical; M=business; L=prototypical/experimental
- Size / Complexity where H=large and complex; M=moderate; L=small and relatively simple

Lifecycle Process Criteria	Waterfall	Incremental	Iterative/Spiral	Evolutionary	Agile/XP
Maintainability	H	H	H	L	L-M
Application Domain	M, H	M, H	M, H	L, M, H	L
Size / Complexity	L, M, H	M, H	M, H	L, M, H	L
Uncertain Requirements	L	M	M	H	H
Progress Visibility	L	M	M	M	H
User Involvement	L	L	M	M	H
Requirements Volatility	L	L	M	H	H
Urgency	L	L	M	H	H

As a sanity check, let's validate this table by locating the key selection criteria or "drivers". The following synopses are written with reference to the entries in this table.

Waterfall: The waterfall model encourages thorough requirements development and design, and formalizes milestones, documentation and deliverables. This process is typically used for safety and reliability critical systems where lifecycle maintainability is a high priority, the systems tend to be large and complex, and the requirements are fairly well understood - at least in terms of general functionality and scope. Waterfall is still very valid for moderately critical and both small and mid-sized projects provided the requirements are well understood.

Incremental: This model essentially scales up the waterfall process for larger or more critical projects with a need for high maintainability. Incremental delivery allows for a degree of requirements uncertainty since poorly understood requirements can be pushed off to a later increments and product releases. Incremental development also offers positive visibility into progress at each release. Observe, however, that management and technical processes must be able to modularize the project to be able to run such a project without losing control. This implies devoting efforts to baseline as much of the requirements as possible and to stabilize foundation components and the most critical architectural modules as early as possible. In comparison to the waterfall, this will increase both technical and management overhead, will yield longer schedules,

and increase overall project costs. However, larger projects can be tackled with this approach.

Iterative / Spiral: The iterative model (advocated by the Rational Unified Process and others) derives many of its properties from Boehm's spiral model – so I am considering them together here. In any case, the iterative lifecycle process appears to be the most flexible and middle-of-the-road alternative. This model introduces the notion of continuous risk assessment and is able to incorporate both incremental and evolutionary development approaches within the model. The iterative model satisfies all of the criteria of the incremental and waterfall models while offering better project visibility, incorporating more user involvement, and dealing better with changing requirements. This model may also respond somewhat more quickly to market demands. It is important to take note, however, that the iterative approach carries even more technical and management overhead than the incremental model and this will therefore escalate project schedules and costs beyond that of the waterfall and iterative models when applied to most problems.

Evolutionary: The evolutionary model aims at providing good visibility into the various technical problems encountered as well as encouraging user involvement and customer feedback. However, it is not intended for maintainability or the delivery of operational releases. Rather, this model should be used to explore difficult/risky technical issues and/or to validate uncertain requirements (e.g. UI prototyping). This model is also very useful in experimental or research settings where the products are for more personal use, for example to produce a proof-of-concept prototype. The work products are not considered production quality, at least for large, medium-sized, complex or mission-critical software project where lifecycle maintainability is a priority. However, evolutionary development is a very useful process for front-ending a more structured waterfall or incremental development process. In fact, iterative models like the Rational Unified Process (RUP) incorporate evolutionary development - but they should ensure that their work-products go through adequate design, review and testing steps before release.

Agile/XP: Agile and XP (Extreme Programming) processes can be traced directly to evolutionary development. Like evolutionary development, these models deal with uncertain requirements by requiring high user involvement. [Note: high user involvement is much harder to achieve in large, complex, mission-critical systems as well as less critical medium-sized projects.] Agile and XP take evolving work products towards delivery and release by putting a lot of emphasis on testing and re-factoring. The emphasis of working code over documentation provides enhanced progress visibility over the other models. These processes encourage on-going re-factoring to ensure that the software design continues to be maintainable. But such continuous re-factoring can lead to fragile solutions and maintainability problems for medium, large and more complex projects, especially if documentation is lacking. Hence Agile and XP are only directly applicable to smaller and low criticality projects. However, like evolutionary development, Agile and XP processes could be scaled up by integrating them into a more structured incremental or waterfall process but the overall software

architecture needs to be conceived at an early stage and a plan for producing sufficient documentation for maintainability needs to be incorporated.

Locating the Best Fit

I am hoping that this discussion will motivate readers to adopt a process selection and tailoring strategy for each new project they undertake. The approach I suggest is that you characterize your project by the process lifecycle criteria described above and then find the column (or columns) in that table that most closely matches your project. And be prepared to establish a “combo” process. For example, your overall process could match most closely the attributes of the incremental process, but you can identify distinct modules for development where you could apply Agile or XP processes to yield beta-versions. Of course, your plan should not forget to call for thorough qualification of these beta-versions (e.g. technical reviews, integration testing, proper documentation, customer review) before being released into operation – this is the part that requires buy-in and commitment from both the company and responsible practitioners.

In an upcoming article, I will be addressing the key process areas mentioned earlier in this article. I plan to provide insights and guidance on how to tailor these processes to the current project you may have at hand.

About the Author

Dr. Kal Toth, Associate Professor in Portland State University's Department of Computer Science, is the Associate Director of the Oregon Master of Software Engineering (OMSE) program. He delivers courses on software engineering principles, software project management, software design techniques and the program's software engineering practicum. He has over 25 years of technical, consulting and management experience in large and small technology companies including Hughes Aircraft, Datalink Systems Corp., BC Software Productivity Centre, and the CGI Group Inc.

He may be contacted at ktoth@cecs.pdx.edu or see <http://www.cs.pdx.edu/~ktoth/>.

For more articles visit www.sao.org/newsletter