

Three Problems Overcome with Behavioral Models of the Software Development Process

Bill Curtis

MCC Software Technology Program
P.O. Box 200195, Austin, Texas 78720

Software development processes are usually modeled by manifestations of the software artifact at given stages in its evolution and the nature of the transformations being applied to it during these stages. Thus, the software process is purported to begin with a stage called something like *requirements development*, or in some cases with the even the earlier step of a *feasibility study*. Such models vary the ordering of process stages (eg., prototyping vs. a traditional waterfall vs. incremental building and releasing). Yet, in all such models the software process is bounded by those activities that initiate and terminate the development of a specific software product.

There are three mistakes that can be made if the software development process is analyzed only with models that focus on stages of transforming the artifact:

- 1) the progression of stages through which the artifact evolves gets confused with the organization of the processes through which people develop software,
- 2) project processes that do not directly transform the artifact are not analyzed for their productivity and quality implications,
- 3) the process is treated as discrete rather than continuous in time (i.e., each project invokes a separate process).

These three problems can only be overcome by making a behavioral analysis of software development and the factors that control its productivity and quality. Such an analysis does not replace traditional models of software product evolution, rather they supplement them with much greater understanding of what controls project outcomes. For instance, Figure 1 presents the *layered behavioral model* used by Curtis, Krasner, and Iscoe (1988) to analyze problems experienced in developing large software systems. The three problems described above result in analytic shortfalls at different levels of this model.

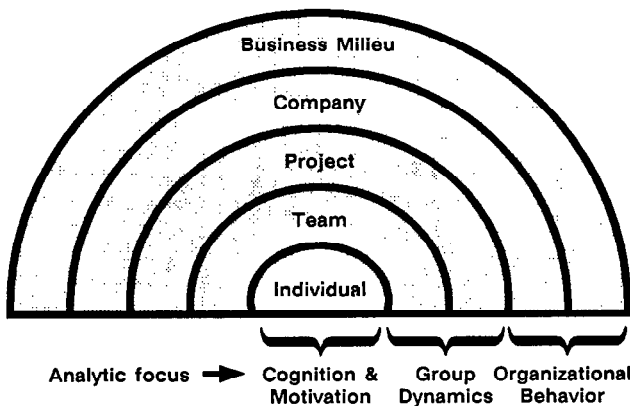


Figure 1. The layered behavioral model of software development processes (Curtis et al., 1988).

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

Confusing project stages with development processes – Since most software engineering textbooks order their chapters to progress through stages of the traditional waterfall model, development is usually depicted as a sequential, top down, balanced process. Many development standards have been written to stabilize this process. One alternative to this model is *prototyping*, in which there is an initial phase of exploratory programming followed by a phase of assessing the resulting program. This sequence of building something and trying it out can be repeated iteratively with each cycle involving the possibility of inserting more structure into the process. At any point in any of these iterative cycles it is possible to shift into the more traditional waterfall process. An interesting application of the cyclic model to assess and control project risk appears in Boehm's (1988) *spiral model*.

These process models for ordering the stages of the software development process are often used to guide the design of software development methods and tools. Since the stages through which the project progresses are ordered either linearly or cyclically, developers are assumed to perform their development tasks in the same order. Thus, the sequence of processes used to guide development at the project level of the layered behavioral model is mapped onto behavior occurring at the individual level. Unfortunately, no one has ever shown empirically that the disaggregation of project level stages had to result in an identical set of staged processes at the individual level.

The grafting of project level stages onto individual level behaviors assumes that mechanisms underlying both project and individual activities are identical. There is currently no theoretical or empirical basis for supporting this assumption. There is no reason to assume that mechanisms governing cognitive processes are controlled in the same fashion as those governing behavior at the project level. Behavior at the project level is affected by an amalgam of factors that involve group dynamics occurring at the team level and organizational behavior occurring at the company level. The project level represents a process abstraction in which management establishes a means of ordering, monitoring, and accounting for what is happening on a project. Traditional software process models provide management with the abstractions necessary to describe behavior at this level.

In research performed at MCC, Raymonde Guindon (Guindon & Curtis, 1988) has demonstrated that the software design behavior of individual designers is better characterized as an *opportunistic process*. These processes are characteristic of those observed in the cognitive literature on planning, and have been modeled in the blackboard architectures used in artificial intelligence research. Guindon has shown that in making design decisions, designers move back and forth across levels of abstraction ranging from application domain issues to detailed design and even coding issues. Decisions to be made at one level of abstraction are compared against their implications for decisions or implementations to be made at other levels. This process is neither sequential nor cyclic, since design behaviors are initiated by the recognition of issues that may occur at any level of abstraction.

If software tools and methods are to provide useful support to software engineers while they are actually performing their tasks, they must be capable of being used in a way that is consistent with how software engineers make their decisions and work with software artifacts. For instance, Curtis, Sheppard, Kruesi-Bailey, Bailey, and Boehm-Davis (1989) have shown how little of the variation in performance is affected by specification formats compared to the enormous impact of individual differences among the software engineers using them. Currently, different levels of abstraction (eg., requirements, software architecture, detailed design, etc.) are usually contained in different documents and are not easily accessed simultaneously in most tools. Next generation design tools must allow software engineers to rapidly switch levels of abstraction in the artifact and support the opportunistic exploration of design decisions.

Ignoring non-transformational processes - Since the software development process is typically described in terms of the transformational stages of the software artifact's evolution, the activities on which these models focus are those that directly serve the transformational process. However, in studying design processes on large software projects in actual development settings, Curtis et al. (1988) found a range of critical problems involving behaviors that are talked about rarely in either software engineering textbooks or models. These activities involve human responses to the situations and conditions under which software development is performed, and involve behaviors at all levels of the layered behavioral model.

The three problems discussed by Curtis et al. (1988) affect cognitive, social, and organizational processes. Since they are based in the behavior of customers and developers, they must be analyzed in behavioral rather than artifactual terms. These problems involve:

- 1) the thin spread of application domain knowledge,
- 2) fluctuating and conflicting requirements, and
- 3) communication and coordination breakdowns.

The first problem recognizes that computer science knowledge alone is not sufficient for building most large systems, since the domain of application can be as complex as the domain of computation. Thus, a software project involves a significant learning component for any project member not already familiar with the application domain. If this learning process is not taken into account in project planning, the time required to build the system will be underestimated. People rarely remember how long it took to learn things before they could actually start doing something. Further, no plan for how to provide the required training will be formulated, leaving software engineers to scavenge for application knowledge wherever they may.

The second problem occurs for many reasons, and frequently because the customer for a computer system is also in a learning process. Far from being able to provide a stable set of requirements, many customers are seeing a range of possibilities unfold before them as developers discuss the implications of the requirements. This learning process continues throughout development, and as a result, these fluctuations perturb the process of transforming the artifact throughout the process.

The third problem involves the enormous amount of time spent by software engineers in communication with each other and the customer. Tools that improve the coordination or collaboration among software project members should have dramatic impact on software development productivity and quality, even though these tools are not actually used to transform the software artifact. The more consistent the understanding of design decisions and constraints are across project personnel, the more consistent the design and implementation will be. Few process models discuss how to organize and manage large development groups to maximize their coordination, but this is as important as managing the transformation process.

Discrete versus continuous process models - Perhaps the most serious shortcoming of most software process models is that they treat software development as a discrete process invoked at the beginning of a project and terminating at the end of a program's useful

life. However, the behavioral factors that have been shown to exert so much influence over software productivity and quality are in many cases not factors that can be managed within the space of a single project.

Boehm (1981) and Valett and McGarry (1989) have demonstrated the dramatic impact differences in personnel capability can have on project performance. These talents are mostly in place before a project begins, and managers scramble to get the most talented people available. However, when queried as to how they are improving the level of talent available to a project, they will reply, "Listen, we hire the best people we can, and there isn't much more we can do to improve the people side of the equation." This attitude is a severe impediment to their productivity on two accounts. First, they assuredly are not doing all they can to hire the best people available. Second, they probably are not doing anything systematic to ensure that they grow and retain the talent they have attracted. The few companies that take recruiting, selection, growth, and retention seriously are noted for their outstanding staff and performance.

A second problem with having a discrete single project model of the software process is that it becomes difficult to argue for the added cost of creating well-designed, highly reliable reusable software components. The cost of tooling up for effective reuse must be paid up front to be reaped out back. If performance (productivity, quality, costs, etc.) is to be accounted for on a single project basis, few managers will volunteer to absorb the costs of providing for future reusability on their watch.

In order to improve the quality of the staff and the availability of reliable reusable parts, the software process must be thought of as a continuous process. It must be reconceived as a process of growing both people and the base of reusable software assets. Rather than squandering the learning that has been paid for so dearly, a plan for a software business will be put in place that projects the growth of business performance in conjunction with the growth of knowledge in the staff and in the component library. Projects are conceived as occurrences in the larger business process that measure the maturation achieved at a particular time through their performance. Thus, the software process is modeled as a set of processes at the company level rather than at the project level. The software process is the continuous growth of the business assets, and their value over time is measured in a series of discrete events constituting software development projects.

References

- Boehm, B. W. (1981). *Software Engineering Economics*. Englewood Cliffs, NJ: Prentice-Hall.
- Boehm, B. W. (1988). A spiral model of software development and maintenance. *IEEE Computer*, 21 (5), 61-72.
- Curtis, B., Krasner, H., & Iscoe, N. A field study of the software design process for large systems. *Communications of the ACM*, 31 (11), 1988, 1268-1287.
- Curtis, B., Sheppard, S.B., Kruesi-Bailey, E., Bailey, J., & Boehm-Davis, D. Experimental evaluation of software specification formats. *Journal of Systems and Software*, 1989, 9 (2), 167-207.
- Guindon, R. & Curtis, B. (1988). Control of cognitive processes during design: What tools would support software designers? In *Proceedings of CHI'88*. New York: ACM, 263-268.
- Valett, J.D. & McGarry, F.E. (1989). A summary of software measurement experiences in the Software Engineering Laboratory. *Journal of Systems and Software*, 1989, 9 (2), 137-148.