On Building Software Process Models Under the Lamppost

Bill Curtis, Herb Krasner, Vincent Shen, & Neil Iscoe
MCC Software Technology Program
Austin, Texas

Abstract

Most software process models are based on the management tracking and control of a project. The popular alternatives to these models such as rapid prototyping and program transformation are built around specific technologies, many of which are still in their adolescence. Neither of these approaches describe the actual processes that occur during the development of a software system. That is, these models focus on the series of artifacts that exist at the end of phases of the process, rather than on the actual processes that are conducted to create the artifacts. We conducted a field study of large system development projects to gather empirical information about the communication and technical decision-making processes that underlie the design of such systems. The findings of this study are reviewed for their implications on modeling the process of designing large software systems. The thesis of the paper is that while there are many foci for process models, the most valuable are those which capture the processes that control the most variance in software productivity and quality.

The Status of Process Modeling

Everyone knows the old story about the man who lost his wallet across the street, but searches for it over here under the lamppost because the light is better. The same phenomenon is true of modelling the software development process. Much of software engineering theory and practice has been borrowed from hardware engineering. The analogies seemed appropriate, and caveats were made where they were inappropriate (e.g., software doesn't wear out). The software life cycle was developed in much the same way, and reflected a model that management and customers felt comfortable with. The resulting waterfall model has been institutionalized in the multi-page foldouts that spread awkwardly from the center of proposals, bearing a thicket of lines connecting boxes filled with illegible 6 point type.

The purpose of process models is to help make software development a more reliable and productive activity. Their focus has been on the management aspects of development: what should be available and when. The assumption underlying this focus is that great leverage on productivity can be achieved by knowing what should should be done by when and by whom. Underlying this assumption is a deeper assumption that the activities comprising any component of the model can be reliably executed. When a given activity is not successfully completed, the process model shows how its effects percolate through the rest of the project. However, if great variability is observed in the performance of many underlying activities, the process model will be less valuable to a manager in explaining what is controlling his outcomes.

On Leaning Against a Waterfall

The waterfall model is one of advanced development leading to a manufacturing process, and it reflects a manufacturing orientation. It specifies interim deliverables and provides checkpoints to ensure that the developing product meets the customer's requirements within the limits of budget and time. Since systems have been built this way for decades, managers feel comfortable expressing their process in these terms. It provides them a basis for assessing risk, predicting outcomes, and tracking progress; in short it provides accountability. For managers, these attributes are the lampposts against which they lean for support and under whose light they search for indicators of their project's status.

Nevertheless, from the beginning of this decade we have heard cries in the literature that the waterfall and related models of software process are inadequate for describing what really occurs in software development. For instance, McCracken and Jackson (1981) soundly denounced the "stultifying" effect of the traditional waterfall model by arguing that it was:

> *"...a project management structure imposed on system development. To contend that any life cycle scheme, even with variations, can be applied to all system development is either to fly in the face of reality or to assume a life cycle so rudimentary as to be vacuous."* (p.551)

Their critique is based on the waterfall's failure to provide adequate mechanisms for managing the inevitable changes in requirements and for involving end-users throughout the development process. The many variations of the waterfall model describe the software process by a set of phases that are terminated by the existence of an artifact. They do not provide much insight into processes occurring within these phases that control a project's outcomes.

The major shortcoming of the waterfall model was that it failed to treat software development as a problem solving process. Not only is the developer attempting to solve the problems presented by the stated requirements and the constraints of available technology, but customers are also trying to solve a problem for which they believe the requirements will yield a solution. Yet, since customers often don't understand the subtleties of their problem, and even more often don't understand the limits of technology, software development becomes a problem solving process involving multiple agents.

A model built on only the end points of the major phases offers little insight into the actions and events that precede the finished artifact (regardless of whether it is design documents, coded modules, or completed tests). Thus, the level of uncertainty and the required learning processes are invisible to those who are responsible for the risk assumed in undertaking the project. When the process is visible, management is in control. When it is invisible, management can only respond to problems, or worse, their symptoms (while hoping to find the source). The manageability of the process is determined by the amount of uncertainty experienced during development. Uncertainty is difficult to predict, because it is an interaction between the novelty of the application or technology employed, and the experience of the personnel assigned to the project.

Alternative Development Process Models

Three important alternatives to the waterfall model have been described in a tutorial by Agresti (1986a) as 1) prototyping, 2) operational specification, and 3) transformational implementation. All three of these alternatives depend on advanced software technology. These three approaches are not mutually exclusive, in that the operational specification is an executable prototype which could be refined through formal transformations. Although these three technologies promise a more productive development process in the future, they have two limitations. First, they are still in their adolescence, and second, they do not handle some of the most crucial factors affecting software productivity and quality.

Recently, Susan Gerhart, Glenn Bruns, and others in our laboratory have completed a year-long evaluation of several major specification and design tools such as Gist, Draco, PAISLey, RML, and Statecharts. Their evaluation included training (often from the technique's designer) and a design experiment using the technique on a standard problem (often the "lift system" that has become a popular vehicle for demonstrating specification and design tools and languages). They concluded that while these technologies are promising, they are, with some exceptions, far from industrial strength. Actual applications of these techniques are sparse, methods for using them are still being defined, and formal language definitions are often lacking. At this stage of their development, it is difficult to determine how effectively some of these technologies will scale up for use on large systems projects.

The second problem, of not handling some of the factors that influence software development productivity and quality, will be discussed in greater depth in a later section. The fundamental problem is that the techniques and tools that underlie alternate approaches to the development process must be substantially more powerful to offload from project personnel the burden of managing the complexity of large systems development. As the size of the system grows from programming-in-the-large to programming-in-the-gargantuan, the factors that control productivity and quality may change in their relative impact. Technologies that work for manageable programs may have less influence on factors that, while benign on moderate-sized systems, ravage project performance when unleashed by a system that must be described as an organizational undertaking rather than a programming project. To realize the challenge placed on our most promising technologies, imagine how to prototype the space station, whose total software component is estimated to grow to perhaps 100,000,000 lines of code. Although prototyping may be useful for answering questions on a piecewise basis during development, it is certainly not the answer at the system level. Congressional auditors would balk at the concept of building one to throw away.

The prototyping and operational specification alternatives to the traditional waterfall model are based on the underlying assumption that what is hard about developing software systems is overcoming the initial lack of knowledge about what the system should do and how it should be structured. Agresti (1986b) proposes a framework for a flexible development process that considers the amount of uncertainty surrounding the functional requirements and the user interface, and proposes variations on traditional life cycle models to accommodate them. Uncertainty is treated as the primary variable determining the need for process alternatives such as prototyping and operational specification. Managing uncertainty suggests that we reconceive the software life cycle as a learning process rather than a manufacturing process. For instance, prototyping is one way to learn about the system to be built; it is trial and error learning.

Process Programming

Recently, Osterweil (1986) has proposed that "software engineering processes should be specified by means of rigorous algorithmic descriptions" (p.1). These "process encodings should themselves be viewed as items of software" (p.1). The goal of process programming is a noble one, to so specify the process of developing software in sufficient depth that some of the chaos is removed from its execution. Process programming draws an analogy between the rigorous specification of a development process, and the rigorous specification of a computer program. "Software engineering activities...can be thought of as being carried out by a complex web of smaller software processes, each of which can be viewed as an algorithmic program aimed at the creation of software products".

If we could accurately specify these activities, then it should be possible to create a software development environment that not only provides exceptional management visibility into the state of the project, but also affords remarkable opportunities to automate the coordination of activities. It is easy to imagine a system that automatically routes the forms through which defect reporting, correction, and release approval are managed.

The problem with the analogy of process programming to computer programming is in the variability of the underlying process being specified. When instructions are submitted to a computer, the programmer believes that the operating system has been sufficiently debugged and that the hardware is sufficiently reliable that the computer will do exactly as instructed. Thus, there is little variability in the response to the process instruction issued. This is clearly not the case when issuing to project personnel the process instruction (or any of its subcomponent process instructions) to design a system. There is tremendous variability both in the means of executing the task and in the results that will be produced based on skill differences, amount of exposure to the customer, etc. Process programming appears to be most valuable for those procedures that involve a complex web of tasks, each of which can be performed within a narrow band of variability. Such a complex web of well understood tasks would be the pre-launch system generation with new data of a large program for the space shuttle.

The examples of process programs given by Osterweil (1986) in Figures 1-7 are procedures that even inexperienced software engineers know how to perform. Being told these procedures is not likely to assist software engineers in performing their tasks with greater efficiency or accuracy, unless their problem was not knowing what to do next. Nor does the existence of these procedural descriptions provide managers with greater insight into impending problems. We would have to specify these procedures if we wanted a computer to execute the task, but if a software engineer does not know them we might question his competence. The coordination of a web of creative intellectual tasks does not appear to be improved greatly by current implementations of process programming, because the most important source of coordination is to ensure that all of the interacting agents share the same mental model of how the system should operate.

The real issue here is that if a process model does not represent the processes that control the largest share of the variability in software development, then it is not helpful in boosting productivity and quality. In fact, there are several dangers in process programming if the procedures represent idealized processes that may not map accurately to actual development behavior. First, management will be deceived by the simplicity of the prescribed process and will not understand what pitfalls are likely to await them. Second, we may automate processes that do not match the way people, and often outstanding people, work. Such premature proceduralization may actually interfere with efficient performance of a complex, or creative, task.

Still there is an intuitive appeal to process programming born of our desire to automate as much of software development as possible. In order to realize this opportunity, we must generate models of design and development processes that possess greater explanatory power than those on which current examples of process programming and similar process models are based. Historically, the specification of process in manufacturing was preceded by an empirical task analysis of the activity. We believe that a much deeper understanding of what really occurs in software development is crucial if models of the software process are to contribute to advances in software productivity and quality. An ancient model of the medical process specified that "IF sick, THEN apply leeches". When the underlying processes of illness were better understood, more appropriate procedures were identified. Similarly, process programming should benefit from more accurate models of the behavior of developing software.

Refocusing Process Models

If software process models are to offer more than illusory comfort to managers that the project really is under control, then we must focus them on something other than phase-ending events and activity descriptions that are useful when there is little uncertainty. It is our primary thesis that this focus should be on the activities that account for the most variation in software productivity and quality. If we can represent these processes, then it may become easier to know how to improve them and use them as indicators of project status.

A corollary of our primary thesis is that these high leverage activities should be determined through empirical research on the software development process. Curtis (1980) and Basili, Selby, and Hutchens (1986) have described how empirical experiments can be performed to gather information on these processes, and Conte, Dunsmore, and Shen (1986) review the software measurement literature. However, the available experiments have not generated the information necessary to analyze processes on extremely large projects. Data gathered from actual projects is necessary for assessing the impact of different factors on the development process and the project's outcomes.

Studies of Large Systems Development

Since the mid-1970s, many corporations have collected data on the productivity and quality of their large system development projects. These data were often used to determine the major factors affecting productivity and quality in a particular programming environment. The factors found to be most significant were then frequently used to drive project cost models. Four of the more important studies were:

1. **IBM Federal Systems Division** - Walston and Felix (1977) analyzed data from 60 projects. Of the six most important factors identified, the first two related to interaction with the customer, while the next four involved project relevant personnel experience and continuity.
2. **TRW Defense and Space Group** - Boehm (1981) collected 63 sets of project information from which to develop his COCOMO cost estimation models. The most important productivity factor was the capability of the personnel assigned to the project. The next three factors involved product complexity, reliability requirements, and timing constraints.
3. **NASA Software Engineering Laboratory** - McGarry (1982) and his colleagues found differences ranging up to 20 to 1 in the productivity of different programmers on Fortran projects ranging up to 20,000 lines of code. This range dropped to 8 to 1 for larger projects. No other factor exercised this level of impact on productivity or quality.
4. **ITT Programming Technology Center** - Vosburgh, Curtis, Wolverton, Albert, Malec, Hoben, and Liu (1984) studied data from 44 ITT projects across a range of business areas. They found that factors under management control accounted for 1/3 of the variation in productivity. However, factors that were not under management control, those related to the business area, also accounted for 1/3 of the productivity variation. No one factor was sufficient to guarantee improved productivity.

Although these studies identified important productivity and quality factors, they generally did not elaborate the process through which these factors exerted their influence on the project. In order to better understand the processes underlying these factors, the Design Process Group of MCC's Software Technology Program conducted a field study of large software systems development to collect data from which we could build models of actual development processes. Our study was similar to these earlier studies in the variety of projects involved, but was different in its emphasis on describing how factors exert their influence during the design process.

Because large, complex systems cannot be designed by any single individual, teams of experts are needed to conduct the design. As the number of agents that must handle and transform design information increases, additional processes begin to impact design decision making. The unique processes created by having the team, project, and company involved in designing a large, complex system all impact the productivity and quality of the project. These processes can be represented in a layered model (Figure 1) in which the size of the project will determine how much influence each layer has on the design process. The efficiency of the information flow between and within these layers will influence, and perhaps determine, its outcome.

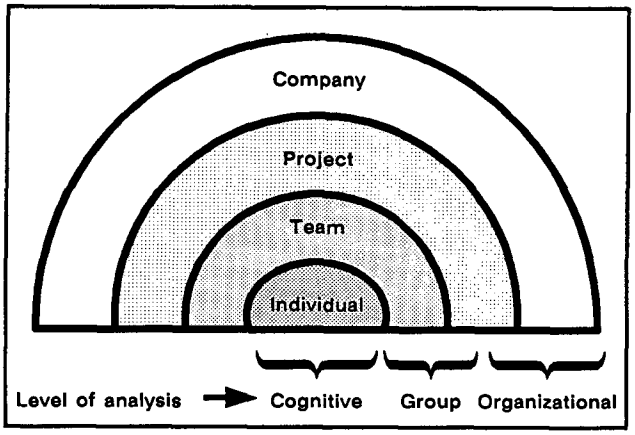

Figure 1. Layers of software process factors

We have now described two types of process models that must be integrated in creating a comprehensive model of the design process. Traditional process models and their alternatives describe how product information grows and is transformed through a series of artifacts over time (i.e., requirements, design, implementation, enhancement, etc.). Design decision-making is an abstract process orthogonal to these models. A different type of model analyzes the behavioral processes of systems development at the cognitive, team, project, and company levels. When we overlay these behavioral processes on the growth process of artifacts, we begin to see the causes for bottlenecks and inefficiencies in the systems development process. The more project managers understand these behavioral processes, the greater will be their visibility into the factors that determine the outcomes of their projects.

Observations of Large Systems Processes

The Design Process Group is developing models of the important, and often elusive, process elements experienced in developing the requirements and design of large, complex systems. The models are

being developed to ensure that our development of an environment for the design of large software systems is problem-driven rather than technology driven (Myers, 1985). An important tenet of our approach is to base these models in part on empirical evidence collected from actual development projects. To this end, we conducted a field study of large systems projects consisting of interviews with important project personnel. This study immersed us in actual development problems and identified key factors for improving processes such as problem formulation, requirements definition, and system architectural design. We focused particular attention on how design decisions were made, represented, communicated, changed, and how they impacted subsequent development processes.

Field Study Method

From May through August, 1986, we visited 19 projects from nine companies ranging from aerospace contractors to computer manufacturers. The original guidelines for selecting projects included the following criteria: 1) projects with at least 10 people, 2) projects that had passed the design phase but had not been delivered, and, when possible, 3) projects that dealt with real-time, distributed, or embedded applications. Most projects selected conformed to some, but not all of these criteria. Occasional deviations from these criteria actually provided a richer set of project types to study. Projects studied varied in 1) the stage of development (early definition through maintenance), 2) the size of delivered system (25K - 100M lines of code), 3) the application domain (system software, transaction processing, telephony, CAE, C³I, radio control, process control, etc.), and 4) key project/system characteristics (e.g., real-time, distributed, embedded, defense contract, etc.).

During our site visits, we conducted hour-long structured interviews with key project personnel such as the systems engineer (whose interviews usually lasted well over an hour), the senior software designers, the project manager, the division general manager, customer representatives, and the testing/QA group. We always interviewed individuals from the first three categories, but were able to interview individuals from the last three categories on only one third of the projects. Participants were guaranteed anonymity and these observations have been "sanitized" so that no individual project or person can be identified.

Our questions focused on *upstream* design activities such as technical decision-making, customer interaction, and intra-project communication processes. Although we structured the interviews to cover certain topics, we asked open-ended questions that allowed participants to formulate answers in their own terms. We encouraged participants to discuss what they thought were important events and challenges during development.

Tape-recordings of the 97 interviews yielded over 2,000 pages of transcripts for analysis. We are taking a two-pronged approach to the analysis of these data. In a top-down approach to the analysis, we began by building models of the important processes we uncovered in the interviews. We have currently worked through nearly half of the interview transcripts to determine how well these models fit across projects, and are making refinements where necessary. In a bottom-up approach, we began by writing summaries of process-related issues for each interview, and then wrote summaries for each project. Notes taken during the interviews were used to guide the preparation of project summaries. The compression from transcripts to project summaries is normally 10 to 1, and we have completed summaries for one third of the projects. The preliminary observations reported here represent a consensus on the issues developed from both approaches and our detailed notes from all the interviews.

Preliminary Observations

A selection of our preliminary observations from the field study relevant to process modeling issues are be presented below. These observations have been selected to provide a view of the software development process based on observations about how the level of application knowledge in both the development team and the customer will affect the development process. These observations will be presented according to the levels of analysis appearing in Figure 1. The impact of factors at each of these levels will depend on the size and nature of the project. On small projects (<10 people or <20,000 lines of code), individual level factors are expected have the largest effect, as was demonstrated in McGarry's (1982) data. On gargantuan projects (>2,000,000 lines of code) organizational level factors should have the greatest impact. However, some of the organizational factors are aggregations of processes mediated at the individual level. For instance, the amount of application knowledge available depends on the individuals assigned to the project. Therefore, under some circumstances, especially those where new applications or technologies are undertaken in large projects, individual level components may continue to exercise leverage on project outcomes.

Individual Level

It is not surprising that we observed substantial differences in individual talents and skills on the projects that we surveyed. Project and division general managers consistently commented on how these differences related to project performance. These observations are consistent with earlier productivity studies at IBM, TRW, and NASA-SEL that found personnel experience and capability to be extremely influential factors (Walston, 1977; Boehm, 1981; McGarry, 1982). The magnitude of these differences can often reach 20 to 1 on some software tasks (Curtis, 1981; McGarry, 1982). Nevertheless, the interviews in this study provided much greater visibility into how these differences get translated into productivity and quality outcomes.

One type of rare expertise that stood out in this study was the *super-conceptualizer* (the keeper of the project vision). Most of the projects had one or two people who were the primary conceptualizers behind the design of the application system. Usually, they were the senior systems engineers who communicated with the customers and made the earliest design decisions. In several of these cases, one of these individuals could be described as a super-conceptualizer (and typically was by others on the project). There were three characteristics that set these individuals apart from other project members. First, they were extremely familiar with the application area (telephony, avionics, electronic funds transfer, etc.). Their crucial contribution was in their ability to map between the behavior expected of the application system and the computational structures required to create this behavior (Figure 2). More often than not, the super-conceptualizers had to integrate several knowledge domains that in unison constituted the application area. For instance, designing avionics software might require expertise in flight control, navigation, sensor-data processing, electronic countermeasures, ordinance, and other areas impinging on the functionality of the aircraft. Yet, super-conceptualizers often admitted they were not good programmers themselves.

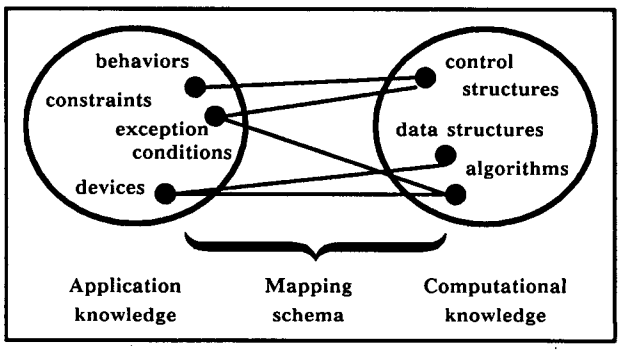

Figure 2. The application to computational mapping

Second, they were skilled at communicating their technical vision to other members of the project, and often spent much of their time educating others about the application domain. Third, they became dedicated to and consumed with the technical performance of the project. In so doing, they frequently became the primary source of coordination on the project and assumed, without formal recognition, many management responsibilities for ensuring technical progress.

The deep application-specific knowledge required to successfully build most large, complex systems was thinly spread through most software development staffs. Most software developers were experts in computational structures and techniques, but were novices at many of the application domains represented in the shareholders' business mix. As a result, software development contains a large commitment of time dedicated to learning. The senior systems engineer(s) who took charge of creating the design usually shouldered the burden of training the staff in the application area, and of assuring that the design and development teams shared the same model of how the system should operate. They also served as advisors to the implementation groups that later worked on project components.

The popular literature on software development argues that no software project should rely on the performance of a few individuals. However, the experiences of many successful large projects indicate why reliance on a few people is more troublesome in theory than in practice. A super-conceptualizer represents a crucial depth and integration of knowledge domains that is arduous and inefficient to attain through a group design process. Under serious schedule constraints, groups may never achieve the level of knowledge integration required to successfully complete the project. Thus, when a project presents serious technical challenges, a super-conceptualizer is often crucial to the success of the enterprise. This type of expertise would be extremely difficult to automate, because expert systems have succeeded only where knowledge could be tightly defined and constrained to a single domain.

Team Level

The formulation, integration, external representation, and communication of the application knowledge required to create the system absorbed most of the project team's time during the early stages of the project. This "knowledge integration" task could be accomplished by either a team or an individual, depending on the size of the system and the number of areas to be integrated. Frequently, the knowledge integration process relied on a single individual (the super-conceptualizer), or on a small, high performance team. This is not surprising given the scarcity of application knowledge experienced on many projects and described in the previous section.

In an idealized model of the team process leading to the creation of a design, the process would begin with each team member holding their own model (a mental representation) of what should be done. These mental models could differ on their representation of the behavior the application system should exhibit, the structure of the environment in which it would operate, the computational model most appropriate for creating the behavior expected of the application system, etc. In the next stage of the group process, individuals sharing similar models would form coalitions to argue for their point of view. In the final stage, the differences are resolved between the conflicting models of various coalitions and a team consensus is reached. This process is similar to the group process for Japanese projects described by Belady (1986).

On most projects we interviewed, the early phases of the project were dominated by a small number of individuals, occasionally even a single individual (the super-conceptualizer). These dominant project members were usually those who knew the most about the application, and they quickly formed a coalition that took over control of the project. Although an idealized model of the team process might include the formation of several coalitions that negotiate the final team consensus, we rarely observed this occurring. Once a coalition was formed, it usually took control of the direction of the project. Thus, if the level of application knowledge required to design a large system was only possessed by a few people, they were able to quickly take control of the emerging design and take it in the direction they believed most appropriate. Competing coalitions rarely formed, because of the speed with which the dominant coalition was able to formulate a design direction. In fact, in some cases the members of the dominant coalition even acknowledged that they had formed a steamrolling group to move the project in the direction they believed it should go. Figure 3 captures what our observations revealed about the idealized model we had hoped to support. This should not be taken to imply that competing coalitions never form, but that their formation is far less frequent across projects than might be expected.

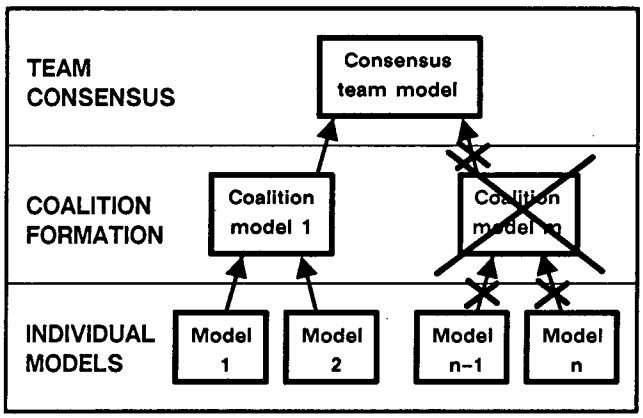

Figure 3. Small coalitions coopt design teams

These observations should not be interpreted to imply that teams are not important during design. In more detailed observation of a design team in our laboratory it appeared that individuals are good at exploring the implications of design decisions in depth, but not in breadth (i.e., considering alternatives). If a broad enough range of concerns are represented among the membership, teams appear to have the complementary characteristic of being able to explore design decisions in breadth by posing alternatives and constraints and challenging assumptions. However, they are not usually good forums for exploring in depth the detailed implications of design decisions. Thus, although a small coalition can take charge of the direction of the design, it can benefit from the challenges offered by opinions which may never themselves gather enough support to redirect the course of the design.

Project Level

Much of what occurs during design is not designing, but the learning required in order to design successfully. The time required for learning application-specific information is buried under the traditional life cycle phase structure of most projects and is often unaccounted for. In some cases this represents the customer's unwillingness to pay for education, since they believe the contractor should already have the knowledge required to build the system. Thus, the time required to create a design is often seriously underestimated, since estimates are usually only based on time spent in creating a design, rather than in becoming educated in the application domain. This phenomenon may also be true for other life cycle phases and activities.

There are several phases of design that are hidden below the surface of most software life cycle models. The first such component

is the process of abstracting the essence of the application behavior. This component requires a thorough understanding not only of the behavior the customer wants out of the application system, but also of the environment the system must operate in. Although customers are able to describe the rudiments of the behavior they desire in the end product, they often have difficulty envisioning the interactions of various parts of the system or of the system with its environment. Nevertheless, these scenarios become crucial in making design decisions at many different levels of the system's decomposition.

It was interesting to observe that much of the information about the application and its environment needed by designers to capture the essence of the application behavior had been generated by customers in discussing the system they wanted. However, this information was not normally captured in any formal way and was usually abstracted out of requirements documents. Thus, the designers had little more than the obvious scenarios of application behavior, and were unable to see the subtleties the customer would ultimately want in the system. In some cases, there were classified documents containing this information, but the software designers were unable to obtain them because they were not considered to have a need to know. The need for prototypes to collect customer reactions to the system's functional behavior might be lower if the information that had always been available among potential users had been captured and used even informally. That is, most projects spend tremendous time rediscovering information that had, in many cases, already been generated by the customers.

Another hidden phase is the selection of a medium for representationing the application's behavior, its environment, and the computational model of how the software would control the system. If the current application was not strikingly different from ones previously developed at the site, then reuse of previous representational formats eliminated this phase altogether. However, when the challenge presented was substantially new, many super-conceptualizers refused to be coerced into using a previously selected representational format. With the right choice of representational medium, the job of communicating system structure and behavior to the rest of the project team was simplified. Issues were usually resolved in the format of the representation selected.

During the educational phase of design, the primary designers of the system ensured that all other team members came to a common understanding of both the desired application behavior and the computational model to be applied in developing the system. If executed effectively, this educational process ensured that the hundreds of programmers and designers staffing a large project were all working on the same system. The senior system engineer(s) usually took charge of this process, and often ran design meetings in a manner not unlike a graduate seminar. *The extent and importance of the educational process during design cannot be overemphasized.*

The number of unresolved issues that carried through the design process was a serious concern for systems engineers. On numerous projects they lamented the lack of available tools for capturing issues and tracking the status of their resolution. The number of issues raised and the proportion remaining unresolved may be the most valuable leading indicator available to management about the progress of the system. Issue management is one of the few opportunities to provide management visibility into the intellectual progress and conceptual integrity of the design. The failure to resolve these issues frequently did not become obvious until integration testing.

It is tempting to conclude that the best prototype is a failed development project. We observed several highly productive projects that had risen from the ashes of a failed project. In these cases it appeared that the failure had allowed the senior designers to immerse themselves in enough of the application and computational problems of the system, that they now understood the essence of the problem. A rapid prototype which failed to expose the designer to the level of application and computational problems that were experienced in a failure may not provide the level of information needed for building a successful system. That is, the designer frequently fails to prototype the system to sufficient depth for the subtle problems of the system to present themselves. Prototypes represent experiments, and like any experiment, the amount of preparatory work to identify the specific information to be obtained from it will determine the value of the results achieved.

Company Level

Belady (1979) and Brooks (1986) argue that change is inherent in software projects. Internal company environments change due to new organizations, technologies, personnel, policies, procedures, etc. External environments change due to economic factors, market conditions, political events, etc. Change stimulated from within the development organization appeared easier to manage, because the parameters of its effects are easier to determine. External change was more perplexing, because the context for technical decisions had changed at the same time that the project was struggling to maintain consistency among the decisions already made.

The problems most frequently cited with *requirements volatility* were of eliciting requirements from customers (even if the customer was an internal organization) and the evolutionary processes these requirements go through in the minds of customers throughout development. Although requirements volatility would seem to be a process better represented at the project level, many of the complaints expressed in the interviews indicated organizational processes that are more appropriately analyzed at the company level. Some examples of specific problems we observed were:

- the marketing department and the development team talk to completely different sources for customer requirements (and these sources may have conflicting needs),
- marketing requests a feature because the competition has it, without understanding the impact of the feature on the existing system's performance,
- the current functionality becomes inadequate because of competitor's products entering the marketplace,
- the customer completely reinterprets the role of the system within the context of other systems with which it interacts or reorganizes his operations and needs different behavior from the system,
- the requirements are tailored to the first customer who places a large order, even though it is known that future customers will want a different configuration or set of features,
- access to the real users is denied because the paying customer is both geographically and organizationally removed from the users of the system,

On large projects the customer interface must be treated as an organizational communications issue. The customer interface is subject to miscommunication because it is at the same time both too simple and too complex. This interface is too simple in that the opportunities for developers to talk with real users are often too limited. It is too complex in that it is often cluttered with different organizational components communicating about their particular concerns, not all of which involve the requirements for the product. In no instance did we find a single point of contact to which the development organization could look for a definition of system requirements. Even in the case of commercial projects, the development organization would find marketing requesting one set of features for a product, but would find a different set of reactions from discussions held from current customers at a users' group meeting.

Often the largest problem in managing a government contract is

in the complexity of the customer interface (Walston & Felix, 1977). This interface usually includes many different agencies with different agendas, each representing themselves as "the" customer. Figure 4 depicts a typical set of customers to interact with when contracting with the U.S. Department of Defense. These customers include senior DOD officials who championed the system, the contracting and procurement office, DOD operational systems engineering, the actual military operators of the equipment, the Independent Validation and Verification (IV&V) contractor, etc. Further, the development organization may itself be a "customer" for the components built by its subcontractors. Multiple customers also contribute to the requirements volatility problem described earlier. In this case, management must negotiate the proper prioritization of requirements among the various "customer" groups.

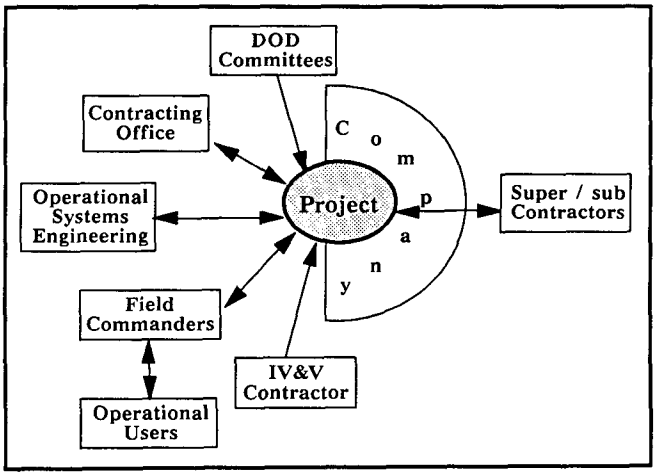

Figure 4. Customer interface on DOD contracts

Most projects had a variety of organizational mechanisms to clarify customer requirements. Occasionally different parts of the same corporation competed to define system requirements. In several cases, we observed that the requirements, and often even the understanding of the product, varied between strategic planning, marketing, standards, and product planning groups. Consequently, conflicting messages were sent to the development organization to be sorted out. The resulting uncertainty about system features often resulted in incorrect decisions about design tradeoffs.

Although better tools were needed to support the analyses required for many technical decisions (performance being the primary issue), it was the complications added by non-technical constraints that generated the most complaints in the interviews. Most professionals felt competent to analyze and resolve hard technical problems, but often found the added complexity of non-technical constraints overwhelming. They found it difficult to perform tradeoff analyses due to a lack of rigorous (and stable) criteria about non-technical issues.

Conclusions

The Orientation of Process Models

Software process models serve many purposes. The waterfall model serves the need for management accountability. Recent alternatives to the waterfall model indicate that the level of uncertainty on most projects makes a strict top-down execution of a development process difficult. However, these models are often based on advances in technology that we believe are still in their adolescence. We have been concerned that there are processes that exert great control over project outcomes that are not represented effectively in existing process models. The distinction we are making is in prescriptive vers descriptive models.

Current software process models are prescriptive in describir how the software development process should proceed. However, the variability of results within a process phase is large, then greate visibility into the activities and events underlying the process needed. We are suggesting a different type of process model for rep resenting software development at a deeper level of understandin than has been characteristic of most management accountability mod els. These models will initially be descriptive. However, they shoul provide empirically-based recommendations for tools and method that are needed to augment the development process. If new technol ogy does not affect the factors controlling the greatest variance in pro ject outcomes, its benefit will not show up in productivity and quality results.

We are building descriptive process models to guide our selectior of the most effective technologies for a software design environment In order to use process models for this purpose, they must describe what project personnel are doing and how this could be improved Management accountability models *do not* give us this insight. The alternatives to the waterfall model offer means coping with some forms of uncertainty in the design, however, they do not provide insight into many other factors that control project outcomes on large systems development. Better descriptive models of the actual processes occurring on large projects can guide managers to investigate phenomena that provide earlier insight into factors that control their outcomes. As we achieve a deeper understanding of the actual processes occurring on large projects, we hope to transform descriptive models into prescriptive ones.

Implications from Our Observations

The conclusion that stands out most clearly from our field study observations is that the process of developing large software systems must be treated, at least in part, as a learning and communication process. At the individual and organizational levels these observations emerge from the large impact that a small subset of the design team can exercise very early in the design process over the rest of the project, because of their superior understanding of the application and its environment. As a result, a large part of their role during the design process is to educate other members of the project team about the application, and to ensure that they come to share a common model of how the software system should operate. The crucial processes that should be modeled at this level are personnel selection, assignment, education, and communication. Although these processes seem obvious, few process models include a recognition of the educational factor in design.

Further, the frequent organizational separation of systems engineering from software development--occasionally even reporting into different vice presidents--inhibits the ability of the application experts to communicate their knowledge to the project team throughout the course of the project. These organizational structures are based on the assumption that the artifacts produced by systems engineers are sufficient to convey all of the information the software developers will need to complete their assignments. We believe that this is not the case and that a process model needs to address the need for constant technical communication between the various engineering groups, especially as the model may account for education on the application or new technologies being employed. For instance, these organizational structures are not designed to enhance communication among engineering groups (hardware, software, and systems) about how feedback about implementation problems should affect system design (e.g., when code growth becomes problematic because of hardware limitations). Process models need to imply recommendations for the organizational design best suited for supporting large systems projects.

At the team and project levels we have seen the development of informal roles which, although not represented in the Work Breakdown Structure, can be crucial to project success. Many of these roles center around intra-project communication. The communication model suggested by existing models of software development is one to support management accountability. It identifies who provides what formal deliverables, reports, etc. to whom. What is missing from this model is the vast amount of communication necessary to ensure that all members of the project (so much as is possible) share the same understanding of the design and its implications for different system components prior to their implementing these decisions. Formal organizational structures are usually designed to report upwards rather than horizontally. Communication systems based on process programming will be helpful in routing formally defined documents, reports, and other artifacts. However, information in formal reports usually reaches its audience well after it was needed for making additional design decisions. One focus for coordinating the information flow within a project would be to augment the informal communication network with better coordination tools (Krasner, 1986).

At the project level the desire to have a process model that provides management accountability inhibits designing a process that most managers will admit is the way it really happens. Although it is popular to say that you want to build an initial version to throw away, few projects design their process to accommodate this approach. The hallmark of management accountability is to assume that you know what you are doing and to present a set of milestones that indicate you have a clear understanding of how you will reach your objectives. The inclusion of a prototype into the process is an admission that there is a learning component involved. However, few process models provide for the level of uncertainty and change experienced on most projects. To handle uncertainty more effectively, process models need to include components for exposure to and modeling of the application and its environment by more than an initial team of system engineers. Important design decisions made late in the design process need information about the application's behavior and environment that was frequently not anticipated early in the specification process. Prototyping is one approach to this problem, but an approach that varies in importance with the amount of information that designers and implementors share about the application domain. Design is in part an educational process, and trial and error learning (prototyping) is only one approach to education, albeit a powerful one.

Prototyping and other alternative models attempt to handle the uncertainty that attends the technical issues within the project. However, most process models are not designed to account for the level of perturbation that results from changes in the strategic objectives of the company (if they were ever crisply defined for the project) or by the changing conditions in the customer's environment. In particular, most process models do not provide guidelines for handling the often Byzantine customer interface experienced on many large projects. Some customer interfaces are so complex that the process of managing them is one of leading negotiations among the various factions within the customer environment. That is, negotiation precedes specification. Few process models handle the negotiation process that not only should precede specification, but attends many customer's torrent of requests for changes.

Summary

Existing models of the software development process do not provide enough insight into actual development processes to guide research on the most effective development technologies. From observations gathered on actual projects, we have concluded that processes such as learning, technical communication, negotiation, and customer interaction are among those crucial to project success which are poorly provided for in most existing models. The observations presented in this study have been offered not to denigrate the importance of management accountability models. To the contrary, we have tried to indicate some important management tasks such as negotiaion that are not accounted for. Our purpose is to indicate the limitations of current process models in enlightening our understanding of software development and in guiding our research on development methods and environments. We are not trying to tear down the lamppost, we just want to build one across the street where the search for one's wallet may be more fruitful.

References

Agresti, W.W. (1986a). What are the new paradigms? In W.W. Agresti (Ed.). *New Paradigms for Software Development* (Tutorial). Washington, DC: IEEE Computer Society, 6-10.

Agresti, W.W. (1986b). Framework for a flexible development process. In W.W. Agresti (Ed.). *New Paradigms for Software Development* (Tutorial). Washington, DC: IEEE-CS, 11-14.

Basili, V.R., Selby, R.W., & Hutchens, D.H. Experimentation in software engineering. *IEEE Transactions on Software Engineering*, 12 (7), 733-743.

Belady, L.A. & Lehman, M.M. (1979). The characteristics of large systems. In P. Wegner, (Ed.), *Research Directions in Software Technology*. Cambridge, MA: MIT Press, 106-138.

Belady, L.A. (1986). The Japanese and software: Is it a good match. *IEEE Computer*, 19 (6), 57-61.

Boehm, B. W. (1981). *Software Engineering Economics*. Englewood Cliffs, NJ: Prentice-Hall.

Brooks, F.P. (1986). No silver bullet - essence and accidents of software engineering. In H.J. Kugler (Ed.), *Information Processing* 86, New York: Elsevier Science Publishers.

Conte, S.D., Dunsmore, H.E., & Shen, V.Y. (1986). *Software Engineering Metrics and Models*. Menlo Park: Benjamin/Cummings.

Curtis, B. (1980). Measurement and experimentation in software engineering. *Proceedings of the IEEE*, 68 (9), 1144-1157.

Curtis, B. (1981). Substantiating programmer variability. *Proceedings of the IEEE*, 69 (7), 846.

Krasner, H. [Ed.] (1986). *Proceedings of the Conference on Computer-Supported Cooperative Work*. Austin, TX: MCC Software Technology Program.

McCracken, D.D. & Jackson, M.A. (1981). A minority dissenting opinion. In W.W. Cotterman, et al. (Eds.). *Systems Analysis and Design - A Foundation for the 1980s*. New York: Elsevier, 551-553.

McGarry, F.E. What have we learned in the last six years? In *Proceedings of the Seventh Annual Software Engineering Workshop* (SEL-82-007). Greenbelt, MD: NASA-GSFC.

Myers, W. (1985). MCC: Planning the revolution in software. *IEEE Software*, 2 (6), 68-73.

Osterweil, L. (1986). *Software Process Interpretation and Software Environments* (Tech. Rep. CU-CS-324-86). Boulder, CO: Dept. of Computer Science, Univ. of Colorado.

Vosburgh, J., Curtis, B., Wolverton, R., Albert, B., Malec, H., Hoben, S., & Liu, Y. (1984). Productivity factors and programming environments. *Proceedings of the Seventh International Conference on Software Engineering*. Washington, DC: IEEE Computer Society, 143-152.

Walston, C. E. & Felix, C. P. (1977). A method of programming measurement and estimation. *IBM Systems Journal*, 16 (1), 54-73.