Software Project Management Under Incomplete and Ambiguous Specifications

ROBERT B. ROWEN

Abstract—**Large system development and government contracts still adhere to a classical life-cycle approach to software development. A major problem in the classical approach is the completeness and clarity of the user requirements. Some authors have raised the possibility of alternate paradigms being more timely. One such paradigm is the use of prototype software models. Even life-cycle adherents have expressed the importance of iterative modeling and cycling between specification and requirements analysis. This author believes that prototyping is an appropriate approach that can be used as a significant feature of the more formal life-cycle process, with little overall reduction in project control.**

This paper explores three aspects of such a development process. First, the underlying assumptions and the evolution of the current life-cycle management control method is discussed. The differing perspectives of the software designer and the user are discussed. A conceptual framework is proposed that graphically portrays this difference in perspective. Second, requirements are assumed to be ambiguous and incomplete. The contents of a requirements document are discussed with the perspective that requirements will always be incomplete until late in the development cycle. Third, prototyping activities have a primary objective of reducing ambiguity. Different prototyping strategies are appropriate for different phases of the development cycle. An altered life cycle (which includes prototyping as a *formal* part of the process) is used to trace the evolution of the requirements document from ambiguous objective to a system reference document.

Keywords—**Software life cycle; rapid prototyping; project management; requirements analysis; software requirements; requirements documents.**

Introduction

LARGE software systems development projects confront two major obstacles. The first is that projects this large normally are not performed by one or two people. The existence of multiple levels of personnel and large expenditures (computers, salaries, etc.) makes a management process that controls the project mandatory. The second hurdle is overall system complexity. The complexity inherent [11] in such systems places a burden on all written documentation in that it be useful as an unambiguous vehicle for communication.

These two obstacles have led to the acceptance of a model for software development known as the *waterfall* life-cycle development model [7], [22], [33]. This model has been cast in concrete by the government's software procurement practices and standards [12]. The model depicts a step-by-step process for transforming user concepts into code. The process also mandates the testing steps needed to ultimately certify the final product. There is an explicit set of checkpoints, reviews, and documents to assist in the management control of the process.

The step-by-step planning has its roots in engineering project management rather than being a unique child of software technology. Many of the basic steps are found in the writings of the cybernetics and general systems theory authors [13]. The starting point of this development model is the generation of system requirements.

Software requirements tend to suffer from ambiguity. Some ambiguity is a result of the prose descriptions used in the document. In the early stages of a project the total design is not completely understood. This leads to incomplete descriptions as well as inconsistencies.

This paper begins with the premise that requirements will always be incomplete when first received. The objective of the design process should be to gradually make the requirements more concrete as the design becomes more detailed. The paper is organized along three major areas.

A context must be set that exposes the underlying assumptions and origins of the current management control method. This method will be traced from early computer projects to the present, taking note of technology changes along the way. Current thinking regarding prototyping will also be discussed.

The second major area will be the contents of a typical requirements document. Each document section will be discussed from the perspective of why, in practice, the document is frequently incomplete. The detailed discussion sets the stage for seeing how the document evolves during the project life as contrasted to freezing the requirements.

The third section of the paper integrates the two notions of a step-by-step development process and the controlled growth of the requirement documents. A key premise is the use of more than one prototyping strategy during the project cycle. The approach does not eliminate the more common project checkpoints but does alter the timing and expectations at various points of the development cycle.

Evolution of the Software Life Cycle

With the development of stored program computers it became necessary to have a systematic development approach. Many of the programmers were mathematicians and their approach to proofs and algorithms carried over into their computer work. One large defense system, SAGE, was developed in the 1950's. Benington presented a paper that outlined the breakdown and organization of effort for a large SAGE related project at M.I.T.'s Lincoln Laboratory [4]. His

Manuscript received February 8, 1988; revised February 12, 1989. The review of this paper was processed by Department Editor R. Balachandra.

The author is with IBM Corporation, Austin, TX 78758.

IEEE Log Number 8932168.

0018-9391/90/0200-0010$01.00

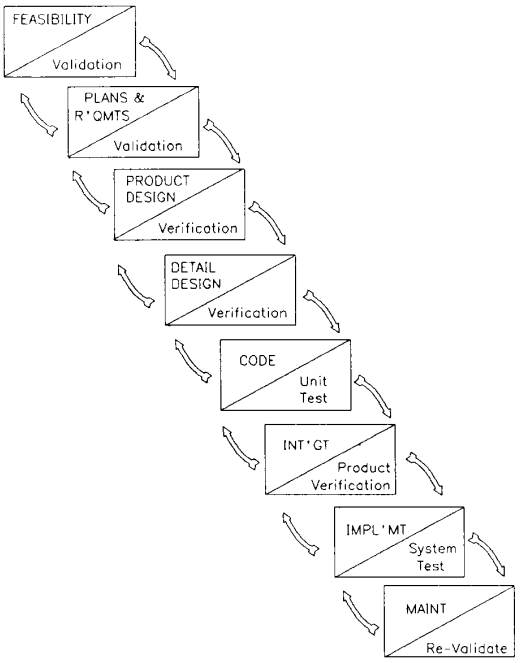


Fig. 1. Classical software development waterfall [7, p. 36]. © 1981 Prentice-Hall, Inc. Adapted and reprinted with permission.

model stressed the feed forward of specification information into (sub)assembly testing (verification) and operational plans were used as part of system evaluation (validation).

The present-day form of the model (see Fig. 1) dates to the early 1970's. The model was presented by Royce of TRW [33] and was very powerfully presented by Boehm, also of TRW, in a classic article in 1976 [6]. The key features of this model were the constant verification steps that attempted to ensure that the previous transformation did not introduce unwanted side effects. Boehm's work on cost estimating [7] used the steps of the model as major categories for cost allocation. By the beginning of the 1980's the life-cycle model was entrenched in the literature as standard practice.

A similar, though a bit more complicated, way of depicting the life cycle is found in the Hughes Aircraft Company's writings [22] (see Fig. 2). The steps are the same (based on DOD procurement) but now the later phases of testing have been made more explicit and the relationship of early design work to later testing phases is clearly accented. A key point to the formal model is that coding is not supposed to occur until the critical design review (CDR) has been successfully completed.

The life cycle is a very attractive model because specific documents, meetings, and deliverables are associated with each step of the process. For the technologists, it is a divide and conquer means of dealing with the complexity of large systems. For management, the model provides known checkpoints and control mechanisms.

While the steps and models were being formalized during the 1970's, another phenomenon was taking place. The electronic industry saw tremendous gains in computing power measured in physical terms such as size, power consumption, and speed. Correspondingly, price was dropping precipitously. Reliability was improving just as dramatically. The result was an explosion of applications and the embedding of computer hardware into many other complex systems. For example, it made little sense, in 1955, to place a 30-ton computer on an aircraft but similar computing power in 1975 was only a 2-oz package [30]. The economics of hardware design were undergoing a significant change.

On the software side of the house, changes were also taking place. As performance increased, tools for program construction became more powerful and widespread. The computer was also put to work managing the complexity of the development process (libraries, compilation, etc.). More significantly, the turnaround for compilation dropped from one or two a day (in a batch environment) to interactive program development. A programmer had to use the "valuable" computer time wisely by being very exact and "paper computing at the desk" between runs during the 1950's and 1960's. Acceptable practice in the 1970's evolved to letting the compiler find the typing errors as the user developed his application.

The net effect was a management control system becoming more accepted and more rigid while the average programmer was becoming less formal. Alternative development methodologies began to surface (see [27]). Even with the alternative approaches, two significant problems are present in both the classical and the new methods. The first problem is to get unambiguous requirements from the prospective user. The second is to have a happy user when the software is delivered (exactly as specified in the requirements).

A key observation is that the first problem (requirements) directly affects the overall performance of the delivered software. As an example, the accuracy of measurements and the response time to a stimulus have a direct bearing on the safety and performance of an aircraft. Ambiguity in the requirements could result in incorrect interpretation of measurements and, ultimately, failure of the system.

The second problem (a happy user) is perhaps a flippant description of a very real problem. Clearly the user will be dissatisfied with incorrect execution (numbers not added correctly). Performance, however, is often a subjective measure. In Lehman's discussion of programming domains, the discussion of E-programs [26] describes the environment of many large scale software projects. E-programs or embedded domain dependent software systems [16] interact with their environment and change the original environment by their operation. One consequence of this is that the user's expectation of satisfactory performance changes as he is exposed to and uses the software system.

A potentially useful addition to the model of the development process, then, is to view the user and software developers as entities having separate trajectories (see Fig. 3). The purpose of a formal control system (e.g., life-cycle methodology) should be to insure that both entities meet at the end point. The span between the two trajectories represents the gap between the user's perception of need and the developer's perception of user needs.

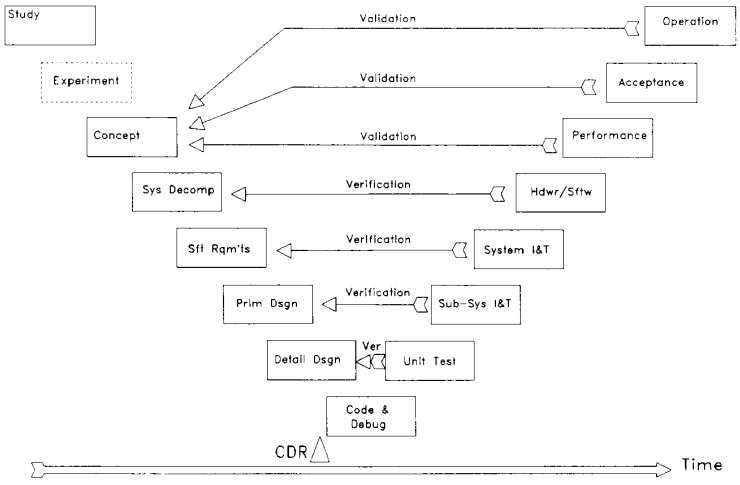


Fig. 2. The "Vee" shaped software development model [22, p. 36]. © 1979 Prentice-Hall, Inc. Adapted and reprinted with permission.

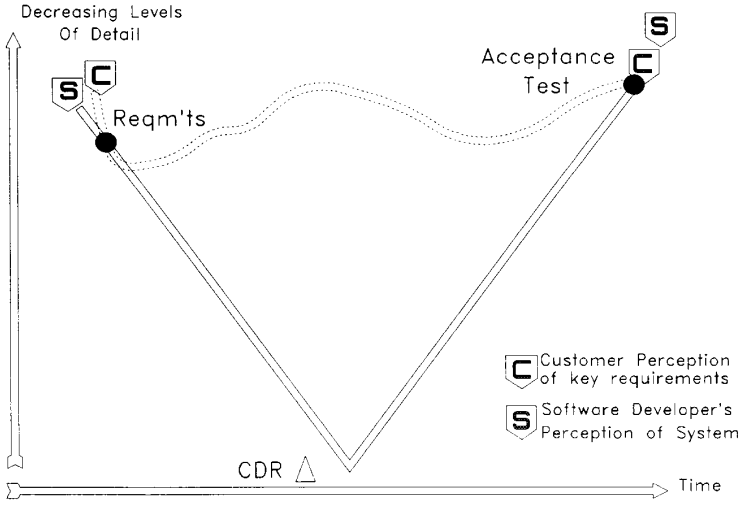


Fig. 3. A framework for software development.

User satisfaction is further complicated by the passage of time between requirements generation and system delivery. As the users gain more insight into the planned environment their goals and expectations change.

Regrettably, the emphasis of the life-cycle steps is management of the budget and the software developers. A tacit assumption is made that by demonstrating *control* of development through reviews and documentation, *controllability* of the user's trajectory is also possible. In the "Vee" of the Jensen and Tonies model (Fig. 4) this control implies that the user will follow the designers to the depth or level of detail present in the critical design review (CDR). My own experience and that of others [18] would seem to reject this assumption.

The Role of the Requirements Document in the Software Life Cycle

The first common point of intersection of the two trajectories is the system requirements document (SRD). In theory, this document should outline the user's expectation of *what* the system must do, not *how* to actually implement the requirements [36]. The intent is for the document to promote communication between the user and the developer. The SRD is the starting point for all subsequent design activity and is also the criteria for validation efforts. There are, however, a few difficulties with this seemingly straightforward notion.

1) Yeh and Zave point out that there are no standards about what should be included in a requirements document [36]. A later (1984) IEEE standard has been published as a guide to good form, but not as a mandatory industry standard [21].

2) Heninger reports that even writing the requirements for a working program with experienced maintenance personnel turned out to be *surprisingly difficult* [20].

3) It isn't clear that the goal of separation of *what* and *how* is an achievable goal even though it is desirable [34].

4) Many prototyping advocates question the assumption that a user can prespecify the details of a system. This is not

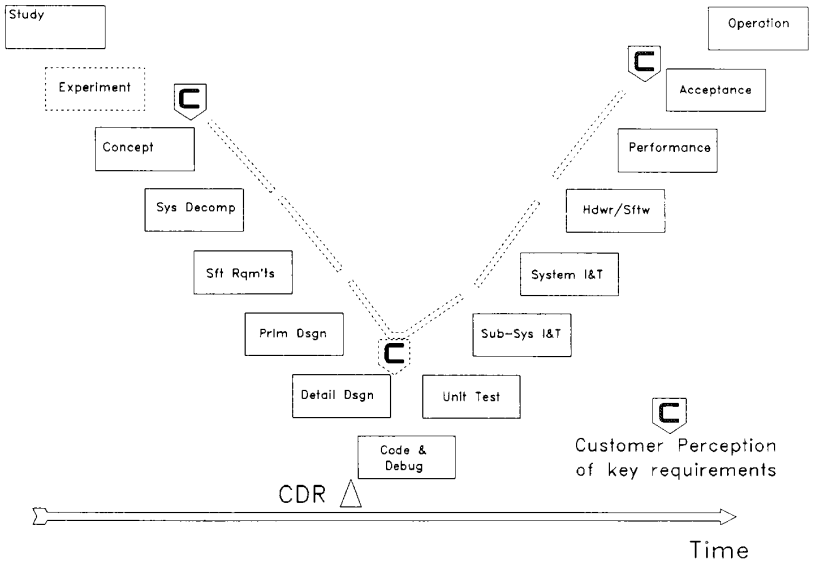


Fig. 4. Customer perception of detail superimposed on the "Vee."

questioning the ability of the user as much as asserting the complexity of the task [5], [11], [15], [24], [27].

There is agreement, however, on the economic leverage of the requirements analysis activity. Wolverton reports a relative cost for correcting a latent error during test and integration as being 36 times the equivalent cost during requirements analysis [35]. This financial leverage is true throughout the life cycle. The sooner an error or misunderstanding is uncovered the easier (in terms of time and effort) it is to correct. Yet one full leg of the development "Vee" is devoted to testing and correction.

It would be highly desirable to have the testing and correction take place as early as possible in the development process. This is the objective of the validation efforts during the life cycle. Initially, however, the only point of reference is the user's incomplete vision of the system. The written documentation will be in the user's terminology and be at a level of detail consistent with the number of decisions made up to that period in time.

A manufacturing system, for example, will have the process steps blocked out and be based on rough throughput calculations. The internal behavior of the blocks will not be specified. The blocks will have names like "DIP Insertion" or "IR Reflow" even though many of the operating characteristics will depend on the vendor or design used to fulfill that block's purpose. This is not because the requestors are trying to separate the *what* from the *how*. The users are documenting the *known* and relying on experience and later experimentation to document the *to be determined.*

The respective design teams, at this intersection point, bring a diverse background of experience and skill to the bargaining table. The initial SRD is, after all, the starting point for a contract for work as well as a system description. It is considered bad form for a software team to presume they *know best* and should be telling the user what is or isn't important. On the other hand, it is unrealistic to believe the user can articulate *all* the functions a software system is likely to need. Both sides have insights that need to be expressed and merged.

A key problem in fostering communication is to have a common terminology. This is one task that the SRD can perform, even if incomplete at the start. The use of block diagrams and visual examples assist in relating the abstract functions and obscure terminology to spatial and temporal relationships. The SRD describes what is currently understood about a particular acronym, function name, or major subsystem. Enos and Tilberg describe this as the environment of the system [14].

In summary, the SRD represents a significant opportunity to provide a focus for discussions and common understanding at a point in time. The development process should assume that the document is incomplete, initially, and view the incompleteness of the document as a measure of the common understanding. Ultimately, the document should evolve to be a reference document. It should not only be an artifact for the ongoing validation that the design and implementation are fulfilling the user's stated needs but should also be the basis for maintenance during the system's operational life.

Accelerating User Feedback Through Prototyping

While agreement on terminology is a first step, the software system is still imperceptible. Software has no physical embodiment like a machine tool. As a result, even a well-written complete requirements document must depend on the mental images the descriptions evoke. This intangible aspect of software is a double-edged sword.

On one edge, physical alterations are not required. Software can be changed quickly. The behavior can be "rewired" to accommodate changes in the interfaces and to incorporate new features that might take months of physical redesign in the

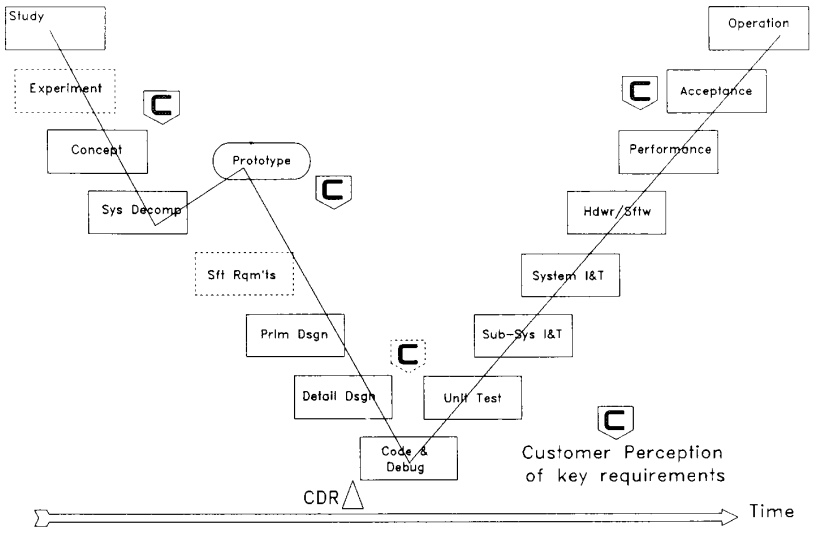


Fig. 5. A prototype altered "W" development model [17]. © 1986 International Business Machines Corporation. Reprinted with permission.

hardware. The software system is often used as a design transient dampening medium between the hardware subsystems.

The other edge of the sword, however, is a dependence on physical results such as reports, screens, or generated control signals to characterize what the software is doing. The software requirements describe the desired behavior in terms of generated signals, screens, etc. The testing of the software will be firmly based on perceived behavior along physical dimensions that approximate the software rather than truly measure the system.

When the engineering and scientific communities are confronted with a complex entity it is natural for them to rely on models to help clarify or visualize the concept. Simple examples of this are wind tunnel models or organic chemistry molecule "kits." A physical embodiment plays an important role in the conceptualization of the system. But models are very often abstractions or representations of the key characteristics, not the entire detailed system. The absence of total knowledge of the details is often a driving force to build the model.

It is the abstraction of detail coupled with the pliability of software that makes prototyping attractive. If the performance parameters are relaxed and the data volume kept small, very realistic models of the desired system can be constructed. An important aspect is that the models can be built quickly (hence the term *rapid prototyping*) using powerful high level languages and hardware.

The literature separates prototyping into three broad categories. These are evolutionary, incremental, and throw-it-away prototyping. The distinction is necessary only in describing the final disposition of the prototype. All three focus on getting user feedback as soon as possible in the development cycle. Not all authors agree that such a split in terms is needed [15], [32].

Throw-it-away prototyping is a consistent use of the term prototype. A feature or system is built, analyzed with the user's help, and then discarded. Documentation, efficiency, and full error handling are normally ignored in the interest of speed of delivery. The prototyping hardware and software are frequently different from the intended delivery environment. The programs described by Boar [5] and Martin [28] follow this approach.

The evolutionary approach differs from the conventional life cycle in that it is intended to produce a production product by the convergence of successive models. Enough development is carried out to enable the user to perform one or more tasks. This results in interim deliveries. Additions and modification are two essential features of the environment. The EPROS system is a current example of this approach [19].

The incremental approach uses an overall internal design, or at least has a framework to bound the process. A closer reading of Royce uncovers the advice to *do it twice* [33]. While at first glance this would appear to be a throw-it-away approach, his intent is to arrange that a version be constructed to model the critical design/operations areas (e.g., based on a design). Brooks advocates building it twice or in a slightly more cynical observation states that you will whether it is planned or not [10]. Today's high level tools have led Brooks to advocate not only twice but many times as you grow a system [11].

Some authors have likened the prototyping activities to bending the "V" into a "W." In Goldberg's discussion of software engineering [17] the process of developing an internal high level design and then prototyping the external screen sequences and interfaces (e.g., behaviors) can be an early evaluation of the system. The end result is an interface specification to build the production system.

In the Fig. 5, the "Vee" has been modified along the lines suggested in Goldberg. Notice that prototyping has the effect

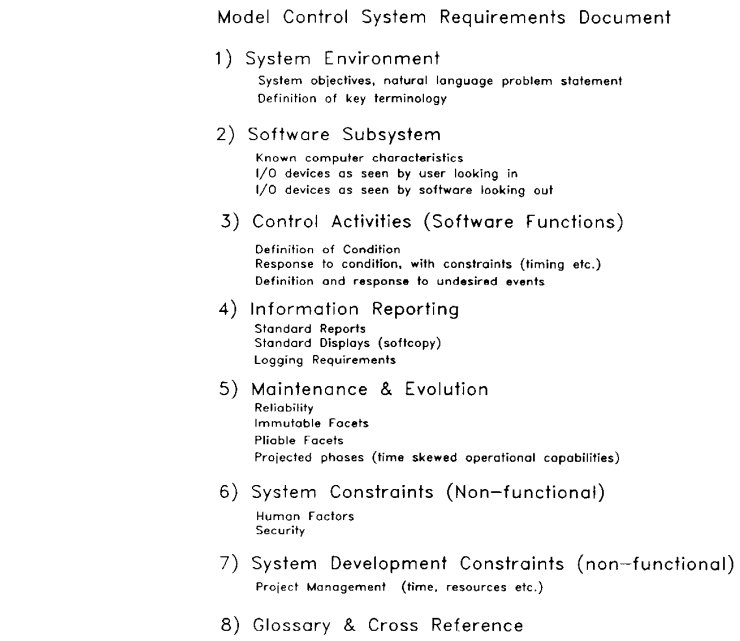


Fig. 6. Generalized table of contents for a requirements document.

of getting the user involved in the process. More importantly, in terms of the trajectory model introduced earlier, the prototype has caused the trajectories to once again intersect.

It appears that both the throw-it-away and incremental prototyping philosophies are consistent with the life cycle's need for user feedback while retaining the intent of a controlled development for the production product. By creating a model we are soliciting user information and attempting to clarify the requirements. The requirements are being tested as behaviors of a physical system, dummy reports, and simplified control examples. It is, in effect, the programming team's way of asking: is this what you were trying to say? The prototype is also an opportunity for the software team members to introduce their own ideas about needed functions but in a physical embodiment rather than a language or abstract description.

Requirements Document Contents

A generic table of contents for a requirements document is shown in Fig. 6. The target environment is some form of control system with real-time elements. This outline is a combination of items from both Yeh and Zave [36] and Heninger [20], along with this author's opinions. The following commentary briefly discusses each section. Some pragmatic observations are made concerning the incomplete nature of the initial requirements. When appropriate, the contributions expected from prototyping are also discussed.

One global comment can be made. The document must address both functional and nonfunctional boundaries and behaviors [36]. These nonfunctional requirements are related to characteristics like security or project management that place boundaries on acceptable solutions. The nonfunctional aspects tend to apply across all the major categories.

System Environment

This section is an orientation section that describes the problem to be solved. The problem involves both hardware and software as well as human operators. One would expect this section to be block diagrams, some preliminary simulation results, and other descriptive material that places the proposed system in perspective. Global objectives or mission characteristics help set the context for later discussions of functions and data manipulations.

As a second subsection, or as part of the narrative, a beginning glossary or definition of key terms would be needed. This would probably be in the jargon appropriate for the task. A later section (Section 8 listed in Fig. 6) provides the detailed reference materials. This first section serves as a level set and common ground for later discussions.

This entire section, at first, would be provided by the user organization. It defines the problem as the user understands it and represents a nonfinancial justification for the work.

Software Subsystem

As the discussion of the environment implies, the software functions are actually only a subsystem of the total problem solution. The specific functions or problem elements delegated to the software would be discussed as a separate but consistent section of the requirements document.

It is possible that the computer has already been chosen for economic reasons or by edict. This is an example of a situation where the *how* (rather than the *what*) becomes part of the requirements. In theory it is desirable to allow the software designers total freedom. In practice, the software may be a subset of a large existing installed base. The software idiosyncrasies of that installed base become important environmental notes for the project.

The software system has two unique perspectives that need to be described. The user sees the system in terms of its external behavior. This may be data logging devices, printers, bar code readers, etc. The anticipated input and output (I/O) devices, and their characteristics should be enumerated. But the perspective of the user is employed to determine what is visible and recognizable.

For the software practitioner, a system's I/O takes on a totally different perspective. Looking out, from within the CPU, the programmer sees channels, registers, mass storage devices, etc. These are capabilities the designer will use to construct alternative designs that drive and service the external devices seen by the user.

The original (preprototyping) document should have the user's "looking-in" perspective well organized and consistent with current thinking about reporting (Section 4 in Fig. 6). If the computer choice is still open, requesting the first portion of this section is meaningless. The software perspective "looking-out" falls between the two. There should be some ground rules for what the design team can expect (e.g., minimum capabilities). But unless a computer has been chosen, part of the prototyping task (and the programmer's creativity) will be spent fleshing out this section.

Control Activities

There are two types of actions that a system is called upon to handle. The first type is planned activities and responses. The second is the unplanned or failure mode activities. Requirements documents normally concentrate on the first since these are the primitives that help describe the system. An enumeration of the inputs and outputs will normally be a good starting set.

It may be impossible to identify internal conditions until after some design has been completed. The possibility always exists that a derived value, through some algorithm, may be replaced by a direct read device (see Fig. 6, section 5). So at best, we can expect a complete list of known input and output lines/sensors. This list will expand and contract as more design takes place during later steps in the cycle.

It is ironic that requirements focus on planned operation but the real complexity of the system tends to be in the error handling and response to unexpected events. Unexpected normally translates into equipment failure or operator error. A programming response may be to report an error or turn a subsection off. This may be unacceptable on a fighter during combat.

This is an area where the prototyping activities can be a catalyst for discussion and design decisions. This type of "what if" requires a lot of iteration between programmers, engineers, and users. Playing "what if" would occur during later phases of prototyping, usually after the initial "scenarios" can be demonstrated. This section should grow as more is learned (i.e., discovered) about the system.

Information Reporting

A response to a condition or event may or may not result in notification to an outside source. An inquiry by an operator may be viewed as an event or the system may have periodically scheduled reporting duties (transaction summaries, etc.). Reports frequently are not real time problems, as compared to an alarm condition or a control point violation.

Ideally the requirements document would enumerate the reports and discuss the contents and calculation of each field. This is a level of detail that will not be known. This is also an area of tremendous human factors impact and personal taste. One would not expect the user to give much thought, initially, to screen layouts. This is clearly a task for prototyping efforts.

Logging requirements are dependent on the definition of failure modes and events of interest. Logs tend to focus on exception conditions rather than standard transactions (a journal tape being the exception that proves the rule). The area of exceptions, as discussed above, is often the weakest part of the conceptual design. This is an area of the requirements that one would anticipate to be sparse in the initial requirements document.

Initially, one would expect the reporting section to be generic. As the design progresses, reports and screens will evolve to standard names centered about their use in the system (i.e., WIP Report, Magazine Contents, etc.). Programmers would like to have standard names, such as SAXM0032, but users tend to truncate the nickname to the "32" report or the "late job tickler." I believe, as these nicknames become understood, the requirements documentation should retain the user name and keep programming conventions in the glossary and cross-reference.

Maintenance and Evolution

Maintenance considerations warrant a separate section. As a number of studies demonstrate, maintenance and operation are 50 percent or more of the total cost of a software project [7], [25]. Ease of maintenance must include planned changes as well as repairs. The work by Lientz and Swanson [25] points out that a large portion of what we call maintenance is actually enhancement or perfective maintenance. Maintenance must be planned and designed into the software; it won't happen by accident [31].

Reliability is an attribute that engineering understands. It is reasonable to expect that the initial SRD will have mean time to failure (MTTF) and mean time to repair (MTTR) targets. This doesn't concede that the targets will be reasonable (e.g., MTTF = 100 000 years) but will give an insight to criticality of the software system. This section should also attempt to define what a failure is. The definition of failure will change over the design cycle but an initial stake in the ground is needed.

Heninger found that a useful device for determining what was subject to change in a system could be extracted by compiling a list of "unchangeable facts or axioms" [20]. The next step was to review the list with the designers and take careful notes of their reactions. If a debate ensued it was a sure bet that the axiom belonged in the possible change (pliable) category. Being able to pinpoint areas that may change is vital information for the design team that partitions software functions into modules [31].

The actual deployment of the system may be planned in multiple stages. There may be valid reasons for adding functions over time. While this is arguably a system project management issue, there are maintenance implications as well.

System Constraints

This section deals with the first of two major nonfunctional categories. The system constraints are factors that broadly impact the delivered product itself. Security, for example, must be considered as a design quality for software routines and will impact the choice of I/O devices later in the design cycle.

Human factors and other design philosophies in this category have a broad scope and cross the more definable boundaries of the prior sections in numerous places. It is fair to assume that the requirements document will give broad treatment to these topics rather than a detailed enumeration of how security or human factors must be handled. These requirements are likely to be stated as objectives and be treated as context data, much the way section 1 in Fig. 6 was goal oriented.

These broad scope topics present an area where the testing team can begin to lay a solid foundation for performance and acceptance criteria. Unlike the responses to events and reports, which have soft meanings initially, these nonfunc-

tional requirements often have corporate definitions or government regulations in place. The testing team is in an excellent position to interpret and refine these regulations. In fact, the test team will probably teach the design team and the prototype will become a vital component in the education process.

System Development Constraints

In sharp contrast to section 7 in Fig. 6, these constraints impact the management of the design team directly. Both Boehm [7] and Brooks [10] have discussed the impact of available time and development methods on the resulting product. The constraints will certainly impact the developed product. But where the constraints of section 7 are technical issues for the product, the constraints of this section would be administrative in nature.

Clearly, this section is just as soft and negotiable as the other sections. Anyone who has participated in a large software development project will vouch for the changeability of this section. Interestingly, prototyping can also have a positive (and negative) impact on this section. A "working" model has motivational and political value. It will be difficult to explain, however, that it is all done with mirrors and can't be shipped.

Glossary and Cross Reference

This is a dynamic but vital section of the requirements document. As the SRD evolves, a new language that represents commonality between hardware and software designers will evolve. I/O signals and reports will begin to take on unique names. Response definitions will begin to tie accuracy requirements and timings to specific I/O lines and channels. An index that relates requirements to design decisions is mandatory.

The maintenance of the code once it has been deployed will, in all likelihood, be by groups not originally involved in the design. Their training will be greatly enhanced if the requirements document can be used as a reference to design decisions and other controlling design documents.

From the preceding discussion it should come as no surprise that a software project manager is faced with two options. If the requirements are complete and detailed, then a lot of the design and modeling has already taken place. Other than turning out code to schedule there is probably little challenge in the project.

On the other hand, if design is actually needed, then the probability of having complete requirements is pretty small. This will require a development approach that accepts incomplete requirements, which is an underlying premise for prototyping.

Classical Formality of Requirements Versus Prototyping

One aspect of prototyping is hard to depict in a diagram. Prototyping is meant to be multiple iterations and not a single step. In fairness this is also true of the waterfall model. Part of prototyping's power is derived from confronting the software designers with the consequences of their decisions "on the terms of the users" [23]. So just like the waterfall model, there is iteration between system decomposition and prototyping. This raises the issue of when to stop or what is the end objective.

It is not uncommon for a requirements document, at the beginning, to have gaps or inconsistencies in the description. The gaps are usually in the level of detail. Operating characteristics are normally given as ranges of values rather than specific operating points. Relationships will not have a crisp timing definition. Some ordering of events will be given but often as a flow diagram with imprecise timings. It would be rare for sample operator displays to be included in the original document, unless an existing process is being automated. Because of the ranges of values and a fuzzy scope of objectives, overlap and conflict of requirements are common.

The requirements analysis phase, under the classic model, tends to be a subtractive process. The inconsistencies, after lengthy analysis and encoding into a formal language, can be weeded out of the document. A consistent data base does not imply a complete set, however. It is the completeness that escapes even the formal description. Once again we are confronted with the assumption that the user will articulate all features.

A prototype approach also does not guarantee completeness. It does, however, present both the developer and the requestor with a visual image and allows other sensory data to create dissonance. Jorgensen [23] relates examples of knowledge by experience as opposed to description. He uses the example of riding a bicycle. Even experienced cyclists are unable to give a full account of what is needed. It is something that you develop a feel for operating. Prototyping can be a very tactile and visual environment that gives the prospective users a feel for the environment. Other mechanisms of cognition can then be brought into use to help judge completeness. The iteration of a prototyping environment tends to be both additive and subtractive.

An objective of the formal approach is to freeze the requirements document. It is not clear that the objective of prototyping should be any different. The fundamental distinction is the *means*, not the end. The prototype approach attempts to first expand the requirements and explore many alternatives before narrowing and freezing the necessary components.

Under an incremental approach or the throw-it-away philosophy there is little intent to deliver the prototype for production use. As noted earlier the "rapid" nature depends on not faithfully producing all the features at real-time performance. Boar succinctly describes this as "no one expects a wind tunnel model to carry passengers" [5]. There is a cautionary note needed. The prototype, no matter how realistic, is still only a model. The scenarios will be contrived and represent only a small fraction of the potential function. The short cuts taken to develop a quick version would be a maintenance nightmare if the prototype was promoted to production use.

I believe judgement is needed to determine the stopping point for both the formal and prototyping approaches to analysis. As will be shown in the next section, the prototypes go through evolution and refinement. But the shift from one phase to the next can be accomplished by specific deliverables.

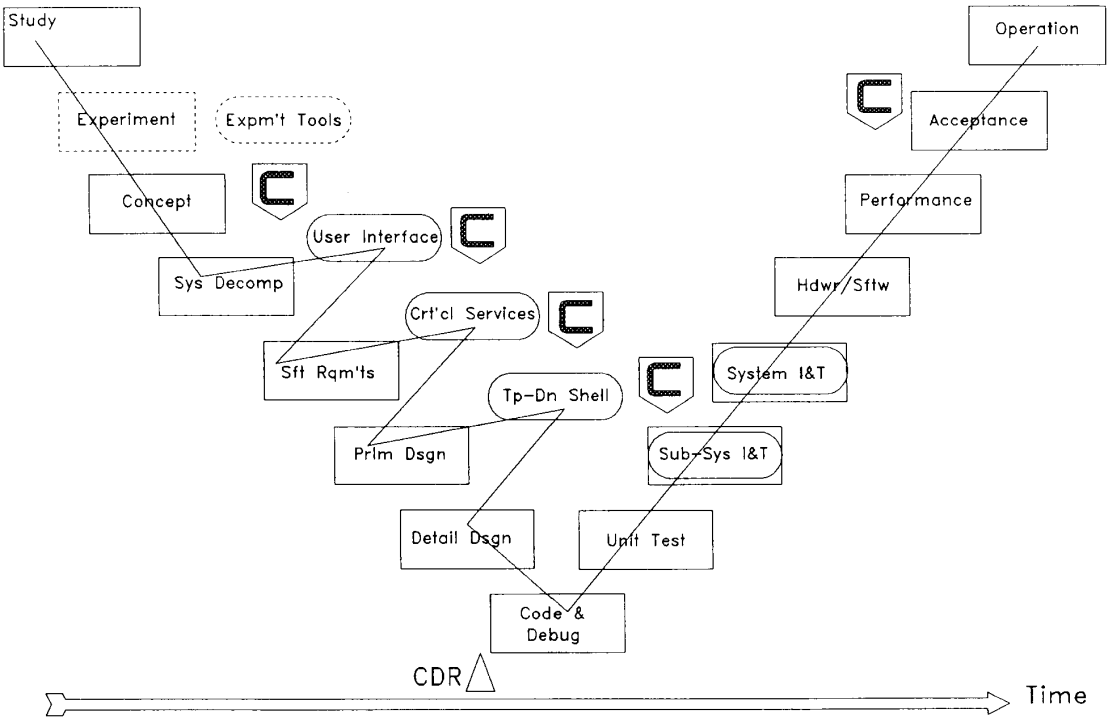


Fig. 7. The "sawtooth" of successive prototype baselines.

Agreement can be reached concerning what features or functions the next prototype must display.

The transition from system decomposition of software requirements, for instance, occurs after both groups agree on a list of functions and have flagged the subset of those functions that are considered critical. The critical functions become the focus of the next phase of prototyping.

The acid test, with any software, is whether the user will authorize work to continue. Lengthy formal documents are often accompanied with political struggles and anxiety over "will the user sign?" Clearly, if the user is not satisified with the prototype (assuming the performance cutbacks are well understood), any more formal design steps would be ludicrous until those issues are resolved.

Integrating the Life Cycle and Prototyping Methods

The throw-it-away and the incremental prototyping strategies are each complementary activities to the classical life cycle. The use of only one of the strategies, however, is not adequate. The primary distinction between the two is that the incremental strategy assumes an overall design. Using an existing design would be counterproductive during the early exploratory system decomposition phase of software development.

The prototyping strategy should change from throw-it-away to incremental as the project matures. The shift in prototyping methods can be timed with the phases of life cycle. As shown by the cartouche items in Fig. 7, a successive group of prototype baselines are created that parallel the development baselines associated with the life cycle. I call this a *sawtooth* development strategy.

This section discusses how the sawtooth creates an environment conducive to creativity and user feedback. This approach is different because it doesn't throw away project controls or familiar checkpoints. It is not a replacement of the life cycle with a prototype methodology. The approach exploits the feedback and user involvement of prototyping but doesn't rely on a single strategy throughout development. The thrust is to evolve the requirements document, a document that is the typical stumbling block during the analysis and decomposition phases of a nonprototyping approach. The early feedback and common visions provided by the prototypes keep both the programmers and users on similar (though not identical) trajectories.

The throw-it-away strategy appears well suited for trying many alternatives and exposing ideas to various groups. The iterations between a concept description and system decomposition seem to fall into the same type of environment. We have seen that screen definitions and reporting descriptions are intially nonexistent. The climate is ripe for the use of report generator tools and display mock-ups to generate user feedback on what is being requested.

During an in-house (single internal development) project it is likely that the software group assisted the engineering organization with support tools for test beds and experiments. These are also throw-away types of programs.

The study of alternatives should settle down to a stable set of user interfaces and a refinement of the software subsystems and reporting sections of the requirements document. The working prototype can then become a working baseline for further development. Overall understanding of the system should be at a point where serious discussions concerning the relationships (timing) of events can begin. In the classic life cycle this is roughly the period between the system requirements review and a system design review.

There would be common agreement, at this point, on what were the critical functions both externally and internally. For example, a particular search function or a transformation path might surface as a time-critical item. If the Pareto principle is accepted, then only about 20 percent of the proposed system will actually require strong control.

The next prototyping iterations are aimed primarily at narrowing the ranges of control values. The result will be alterations to the earlier prototype to reflect what has been identified as the key functions and elements. Notice that during the early steps the operator interfaces took the center stage and some representative functions were modeled. Now with some system experience because of the prototyping efforts a true selection of critical functions can be examined. I have labeled this the Critical Services prototype block in Fig. 7.

Armed with this information the preliminary design should rapidly take form. A critical shift in prototype strategy should take place. With a preliminary design the notion of iterative prototyping becomes realistic, since it requires a consistent overall plan. There would appear to be a fine line between a working prototype and the shell normally present if a top-down system integration approach was being followed. In fact making the transition from an incremental prototype into the subsystem test leg of the "Vee" appears to be a natural transition. Some of the advantages of a top-down approach, such as the morale improvement from always having something working [11], appear to be achievable by the proper staggering of prototype baselines.

Returning for a moment to the trajectory framework, it is reasonable to argue that some guidance has been given to the user's trajectory by keeping a working model in front of them. As depicted in Fig. 7, the irregular "W" has become a sawtooth figure with the points reaching out to capture the user with questions pertinent to the user's own interests. After getting a feel for the screens and overall interface we have started to ask detailed questions about relationships.

With the relationships stabilized, we begin to model and demonstrate the relationships flagged as critical. The preliminary design comes to life as part of the external behavior of the top-down shell. The users may or may not descend that extra notch to the detailed design, but we have kept a constant watch on their perceptions of the design objectives. We have also set up the user to be a participant in the testing phases, long before acceptance testing.

Planning and execution for the testing team follow a similar trajectory. Instead of always working with a paper model or description of the requirements they have a working model.

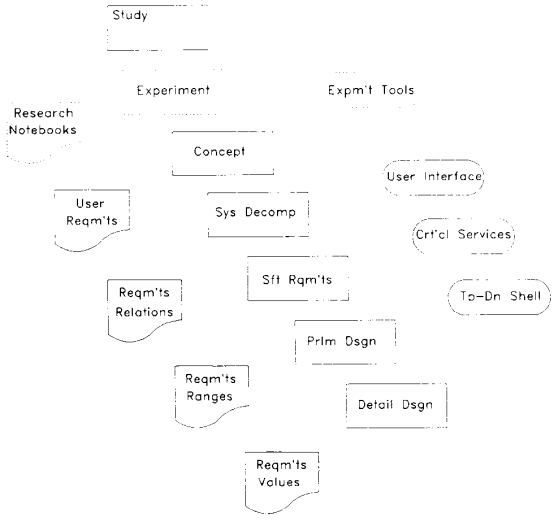


Fig. 8. The requirements document evolves over the life cycle.

Perhaps they are given control of the baseline models from the beginning with the objective of making the top-down shell the test bed for subsystem testing. An independent test team is normally used for integration testing. This team could also assist in moving the top-down shell to real hardware (if different). Getting the necessary testing "probes" installed could take place during the code and unit test time frame of the life cycle.

Document Maintenance Over the Entire Life Cycle

The requirements document has undergone some evolution as well during this downward leg of the cycle. As Fig. 8 displays, the requirements document goes through successive changes to reach a reference document state. The successive steps provide, in a way, a rolling validation effort with continued user feedback (because of the prototypes).

From the original incomplete user requirements the document is augmented to identify screens, reports and functions with the aid of user feedback and a throw-it-away mode of prototyping. The identification of event relationships is complementary to system decomposition and activity focusing on critical services.

Data and experience with critical functions assist in the setting of ranges for events. Subsequent design helps to fix the ranges as values and insures consistency in function descriptions and responses. At this point the requirements document is complete and a complement to detailed design, software specification documents, and coding activities.

Management Reviews and Orientation

The management checkpoints and documents have not changed significantly when compared to the classical waterfall model. It can be argued that these throw-it-away and incremental activities were taking place but without formal sanction. So, in a way, nothing totally new is being proposed.

A shift is occuring, however, in the timing of reviews. The freezing of some documents is being postponed. The testing team may need an advanced staffing profile. There is certainly greater user interaction, which can be time consuming for both groups.

There is a significant orientation change needed. Rigid controls and formal change control procedures may need to be delayed, perhaps until after the preliminary design. The reports, screens, and housekeeping routines may need a much looser application of controls than the level of control needed for the critical services.

We still need to emphasize that the prototypes are only models and shells. It should be comforting to management that visible progress can be demonstrated. I doubt if a higher stack of paper at each review is that reassuring. With a working changeable model, the user sees the software organization as *responsive* and sees them more frequently, perhaps even perceiving them as team members.

Conclusions

The dramatic increase in the performance to price ratio of computing hardware has made the development of powerful high level languages and programmer tools a viable economic pursuit. These productivity improvements, while not as extreme as the hardware improvements, have weakened the underpinnings of the classical software life cycle. Alternative approaches, such as prototyping, have been employed with success.

A sawtooth process has been proposed that creates a hybrid of the classical life cycle and a combination of prototyping strategies. One key aspect has been the use of the prototypes to influence the user's perception of the design (e.g., the trajectory). This, coupled with a top-down integration plan, it is argued, would still allow a controlled software development project. Such an approach would have the effect of getting earlier user feedback (e.g., validation) but also serve as an ongoing focus for the user's evolving perception of the system. The integration of the prototypes and the shifting of life-cycle events has been examined. No loss of management control is created by using this approach.

This paper has also examined a generic requirements document. The author has offered some rationale why the requirements may be incomplete when first received. The sawtooth process incrementally transforms the operating values from a range of numbers toward more precise values. The benefit of using a combination of prototyping development strategies has been discussed as a viable approach for this transformation. This approach addresses a frequent problem, the ambiguity of the requirements document, without forcing a freeze of the document. The frozen document, in the classical life cycle, often needs frequent unfreezing (change control) as the user's perception of needs changes.

References

[1] W. W. Agresti, Ed., "New paradigms for software development," IEEE Tutorial, IEEE EH0245-1, 1986.

[2] M. Alavi, "An assessment of the prototyping approach to information systems development," *Commun. Ass. Comput. Mach.*, vol. 27, no. 6, pp. 556-563, June 1984.

[3] P. C. Belford, A. F. Bond, D. G. Henderson, and L. S. Sellers, "Specifications: A key to effective software development," in *Proc. 2nd Int. Conf. Software Eng.*, Oct. 13-15, 1976, pp. 71-79.

[4] H. D. Benington, "Production of large computer programs," *Annals History Comput.*, vol. 5, no. 4, pp. 350-361, Oct. 1983.

[5] B. H. Boar, *Application Prototyping, A Requirements Definition Strategy for the 80's.* New York: Wiley, 1984.

[6] B. W. Boehm, "Software engineering," *IEEE Trans. Comput.*, vol. C-25, no. 12, pp. 1226-1241, Dec. 1976.

[7] B. W. Boehm, *Software Engineering Economics.* Englewood Cliffs, NJ: Prentice-Hall, 1981. © 1981 Prentice-Hall.

[8] B. W. Boehm, "A spiral model of software development and enhancement," *ACM SIGSOFT Software Eng. Notes,* vol. 11, no. 4, pp. 14-24, Aug. 1986.

[9] R. J. Boland, Jr., "The process and product of system design," *Management Sci.*, vol. 24, no. 9, pp. 887-898, May 1978.

[10] F. P. Brooks, Jr., *The Mythical Man Month.* Reading, MA: Addison-Wesley, 1975.

[11] F. P. Brooks, Jr., "No silver bullet: Essence and accidents of software engineering," *IEEE Comput. Mag.*, vol. 20, no. 4, pp. 10-19, Apr. 1987.

[12] *Defense System Software Development,* Military Standard DOD-STD-2167, June 4, 1985.

[13] N. L. Enger, "Classical and structured systems life cycle phases and documentation," in *Systems Analysis and Design, A Foundation for the 80's,* Cotterman *et al.*, Eds. New York: North-Holland, 1980, pp. 1-24.

[14] J. C. Enos and R. L. Van Tilberg, "Software design," in *Software Engineering,* R. W. Jensen and C. C. Tonies, Eds. Englewood Cliffs, NJ: Prentice-Hall, 1979.

[15] C. A. Floyd, "A systematic look at prototyping" in *Approaches to Prototyping, Proc. Work. Conf. Prototyping* (Namur, Belgium), Oct. 1983. New York: Springer-Verlag, 1984, pp. 1-18.

[16] R. V. Giddings, "Accommodating uncertainty in software design," *Commun. Ass. Comput. Mach.*, vol. 27, no. 5, pp. 428-434, May 1984.

[17] R. Goldberg, "Software engineering: An emerging discipline," *IBM Syst. J.*, vol. 25, no. 3/4, pp. 334-353, 1986. © 1986 International Business Machines Corporation.

[18] H. Gomaa, "The impact of rapid prototyping on specifying user requirements," *ACM Software Eng. Notes,* vol. 8, no. 2, pp. 17-28, Apr. 1983.

[19] S. Hekmatpour, "Experience with evolutionary prototyping in a large software project," *ACM Software Eng. Notes,* vol. 12, no. 1, pp. 38-41, Jan. 1987.

[20] K. L. Heninger, "Specifying software requirements for complex systems: New techniques and their application," *IEEE Trans. Software Eng.*, vol. SE-6, no. 1, pp. 2-13, Jan. 1980.

[21] *IEEE Guide to Software Requirements Specifications,* ANSI/IEEE Standard 830-1984, Feb. 1984.

[22] R. W. Jensen and C. C. Tonies, *Software Engineering.* Englewood Cliffs: Prentice-Hall, 1979. © 1979 Prentice-Hall.

[23] A. H. Jorgensen, "On the psychology of prototyping," in *Approaches to Prototyping, Proc. Work. Conf. Prototyping* (Namur, Belgium), Oct. 1983. New York: Springer-Verlag, 1984, pp. 279-289.

[24] G. B. Langle, R. L. Leitheiser, and J. D. Naumann, "A survey of applications systems prototyping in industry," *Inform. and Manag.*, no. 7, pp. 273-284, July 1984.

[25] B. P. Lientz and E. B. Swanson, *Software Maintenance Management.* Reading, MA: Addison-Wesley, 1980.

[26] M. M. Lehman, "Programs, life cycles, and the laws of software evolution," *Proc. IEEE,* vol. 68, no. 9, pp. 1060-1076, Sept. 1980.

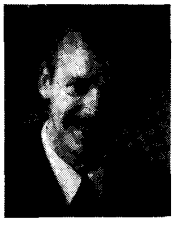
[27] D. D. McCracken, "The changing face of applications programming," *Datamation,* pp. 25-30, Nov. 15, 1978.

[28] J. Martin, *Fourth-Generation Languages.* Englewood Cliffs, NJ: Prentice-Hall, 1985.

[29] K. Matsumura, H. Mizutani, and M. Arai, "An application of structural modeling to software requirements analysis and design," *IEEE Trans. Software Eng.*, vol. SE-13, no. 4, pp. 461-471, Apr. 1987.

[30] J. D. Musa, Ed., "Stimulating software engineering progress, a report of the software engineering planning group," *ACM Software Eng. Notes,* vol. 8, no. 2, pp. 29-54, Apr. 1983.

[31] D. L. Parnas, "Designing software for ease of extension and contraction," *IEEE Trans. Software Eng.*, vol. SE-5, no. 2, pp. 128–137, Mar. 1979.
[32] R. Patton, "Prototyping—A nomenclature problem," *ACM Software Eng. Note,* vol. 8, no. 2, pp. 14–16, Apr. 1983.
[33] W. W. Royce, "Managing the development of large software systems: Concepts and techniques," in *WESCON Tech. Papers,* Aug. 25–28, 1970, pp. A.1 1–9.
[34] W. Swartout and R. Balzer, "On the inevitable intertwining of specification and implementation," *Commun. Ass. Comput. Mach.*, vol. 25, no. 7, pp. 438–440, July 1982.
[35] R. W. Wolverton, "Software costing," in *Handbook of Software Engineering,* C. R. Vick and C. V. Ramamoorthy, Eds. New York: Van Nostrand Reinhold, 1984.
[36] R. T. Yeh and P. Zave, "Specifying software requirements," *Proc. IEEE,* vol. 68, no. 9, pp. 1077–1085, Sept. 1980.



Robert B. Rowen received the B.S.E.E. degree from the University of Wisconsin-Madison in 1972, the M.S.E.E. degree from the University of Minnesota, and the M.B.A. degree from St. Edwards University in Austin, TX. In December 1988, he received the Ph.D. degree in electrical engineering from the University of Texas at Austin.

He has held various professional and management assignments at IBM since 1972. Currently he is an Advisory Systems Analyst in the manufacturing engineering function at IBM in Austin, TX. His research interest is focused on manufacturing software systems, project management, and expert systems. A major facet of this work is software requirements and user involvement in the development process.

Dr. Rowen is a member of the IEEE Engineering Management Society, SME/CASA, ACM, and AAAI.