J. Webster (ed.), *Wiley Encyclopedia of Electrical and Electronics Engineering*

SOFTWARE DEVELOPMENT MANAGEMENT

The process of creating an artifact involves the solution of a constraint problem. The constraints are resources such as raw material and time, and the desired solution is generally a tangible product that fulfills an existing need or requirement. The value of the artifact is measured qualitatively in how well it caters to the existing need, and in its promise to deliver on future need trends.

The dynamics involved in developing an artifact is a function of its complexity, which in turn is dependent on the intellectual resources deployable for its development. The development of a simple product involves an inference engine capable of breaking up the macro problem into manageable subproblems, and subsequently solving each of them in a systematic manner. A problem of small dimensions will yield a range of easily soluble subproblems under this dissection. But as the complexity of the macro problem increases, each subproblem that results from decomposition can get complex enough to warrant the allocation of a powerful inference engine, dedicated solely to its solution. As the problem size increases, the scenario shifts from one that involves a single inference engine concerned with problem decomposition and solution, to one that has a number of independent inference engines operating on separate, yet integrable, subproblems. This shift in form and structure of the solution approach opens up a whole new domain of issues. Under such circumstances, a new entity that coordinates the operation of the various inference engines to deliver a solution within global constraints becomes a necessity. The study of various attributes that define the functions of this coordinating entity with a view to delivering a fully integrated product is called *process management* (1). Extrapolating from this point of view, one may define *software management* as the "glue logic" that seamlessly cements various concentrated software development activities into a composite functional whole, satisfying all requirements without violating resource constraints (2).

During the nascent days of the computer, program developers were a select, highly specialized few, who knew more about the machine than about the nuances of the problem they were assigned to solve. The human–machine interface was so rudimentary that one needed to have spent a considerable amount of time and effort in acquiring competence in putting the machine to good use. But at that time, in the early 1940s, the main objective of computer engineers, such as Atanasoff and Mauchly, was to prove a concept. The development of the *ABC* (Atanasoff–Berry Computer) at Iowa State in 1941, followed by the unveiling of the *ENIAC* (Electronic Numerical Integrator and Computer) at the University of Pennsylvania in 1946, signaled a new era in the history of technology. These were the first resolute steps taken towards firmly establishing the electronic computer as a generic problem solving engine. Barely two decades later, it led to the mushrooming of the multibillion dollar software industry that we see today.

The fundamental stumbling block during the early days of the computer was the highly rudimentary human–machine interface (3). The complexity involved in mapping human problems into simple algorithmic primitives that can be understood by a machine kept the computer from evolving into anything more than a specialized scientific tool. But early research by pioneers at labs such as the Palo Alto Research Center created innovative concepts that made the computer so user-friendly that industrialists saw in it the promise of a marketable consumer product. The computer is now virtually ubiquitous, and supports a wide range of

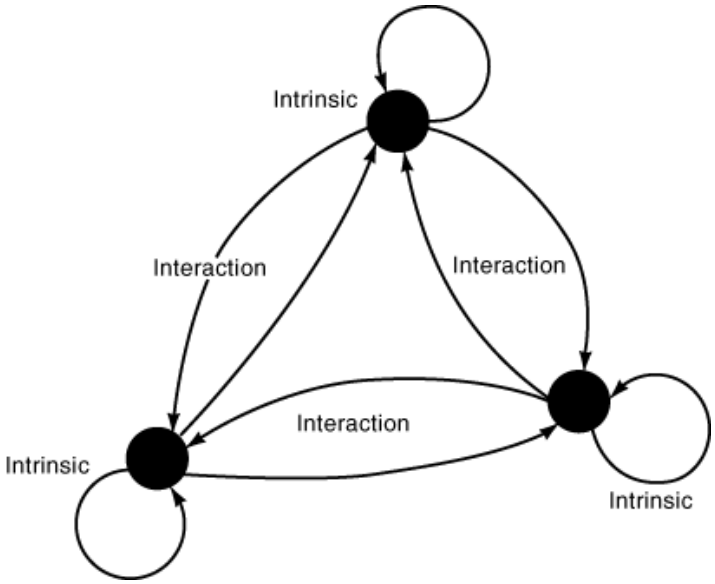

Fig. 1. The interaction complexity within a team. Each individual processes information locally, and may transmit it to colleagues.

applications, limited only by human creativity. A computer, in one form or the other, touches every day of our lives.

Driven largely by the success of the machine in automating repetitive deterministic tasks, the early sixties and seventies saw computers being deployed extensively in defense and in heavy industry. The evolution of the human–machine interface enabled large groups of developers to work as a team, developing software that catered to increasingly complex needs. Strides in hardware technology enabled computers efficiently to store increasingly large programs. Today a fairly functional operating system runs into a few million lines of code, and can involve teams that range from a few to a hundred software designers to manufacture specialized software. The much advertised creation of the personal computer brought yet another dimension to software engineering. The concept of software for entertainment, and with it the possibility of spiraling profit, became a reality. The eighties saw the computer shed its conventional role as a scientific problem-solving machine and become a gateway to information and recreation, a necessity in every household, something as fundamental as television.

The development of large software systems has become akin to a process such as shipbuilding. The issue here is not product duplication as in the automobile industry. The challenge lies in the production of a single custom–made integrated entity that lives up to requirements. Software duplication today, thanks to technology, is virtually a nonissue. And, as in all large-scale systems, development efficiency is a combination of individual skills and the ease with which individual team members share information. As team sizes grow, the complexity of relationships increases as the square of the number n of members—more precisely, as $n(n-1)$. This relationship is shown in Fig. 1. In addition to ensuring a high level of skill and motivation amongst individual developers, an important function of the software manager is to keep the team member interaction complexity to an absolute minimum. Software management, as a discipline, is the systematic study of the ways in which such parameters can be measured, interpreted, and applied to the control of the production process.

The Software Development Life Cycle

Attempts have been made by many researchers to categorize the various phases that a typical software entity transitions through before reaching completion. A software life cycle is defined as "[the activity related to the software during] the period of time beginning when the software product is conceived and ending when the resultant software products are no longer available for use" (4). A software development life cycle (*SDLC*) can be broadly divided into various phases (5), each phase being characterized by a well-defined set of functionally related activities. A model to represent such a life cycle helps developers define their tasks more precisely. It helps software managers track the project schedule and aids in the verification of requirement specification as the project progresses.

Studies in the late 1960s led to the formulation of the *waterfall model*, a well-defined process, with clear milestones. The waterfall model, shown in Fig. 2 is basically an overcautious approach to software development, used extensively in defense projects, with well-defined start and end points.

Software development has been based on the waterfall model or its variations for quite some time. There is a natural tendency among developers to proceed in a highly sequential, linear, and noniterative manner. Designers tend towards perfectionism in trying to make the analysis and design of the product as complete and precise as possible, before even embarking on its implementation. Every iteration, if any, to refine the design is viewed as an indication of an insufficiency in the design. In this paradigm, tampering with the original conceptual design is discouraged, and though designers do iterate, they do so only as a last resort.

The waterfall model does not have a well-defined method of prototyping. The paradigm stresses refining the deliverable from each phase to the highest degree possible before the next phase begins. Such an approach is not usually feasible under some circumstances, especially where the product under development is highly complex and involves several unknowns. The sheer complexity of the requirements specification can render a precise and detailed design impossible. A considerable amount of trial and error is unavoidable in cases where research forms an integral part of the development cycle. The problem with designing state-of-the-art products is that usually the most efficient design is not yet known at the analysis stage. The concept of *rapid prototyping* was introduced to model development efforts that were based on changing technology or imprecise requirements. The principle, according to this paradigm, is to develop a crude version of the final product as quickly as possible, and then subject it to iterative refinement (4); redefining requirements and changing the implementation through each iteration as shown in Fig. 3. Rapid prototyping life cycles have a drawback in that milestones are not clearly defined. As a result, the development cycle tends to *churn*, or iterate a lot more than is required, leading ultimately to schedule slippage. Eventually, the prototype gets shipped out as the finished product, possibly falling short of customer requirements.

Milestones were incorporated into the prototyping approach in an attempt to make the life cycle more trackable. The *fountain* model, used extensively in iterative object-oriented environments, seeks to represent the development life cycle in terms of well-defined phases, as shown in Fig. 4. The various phases defined under this paradigm are:

(1) Requirements study and feasibility report generation
(2) End user requirements specification
(3) Analysis
(4) Preliminary design
(5) Detailed design
(6) Implementation and unit testing
(7) Unit integration and system testing
(8) Code generalization and library generation

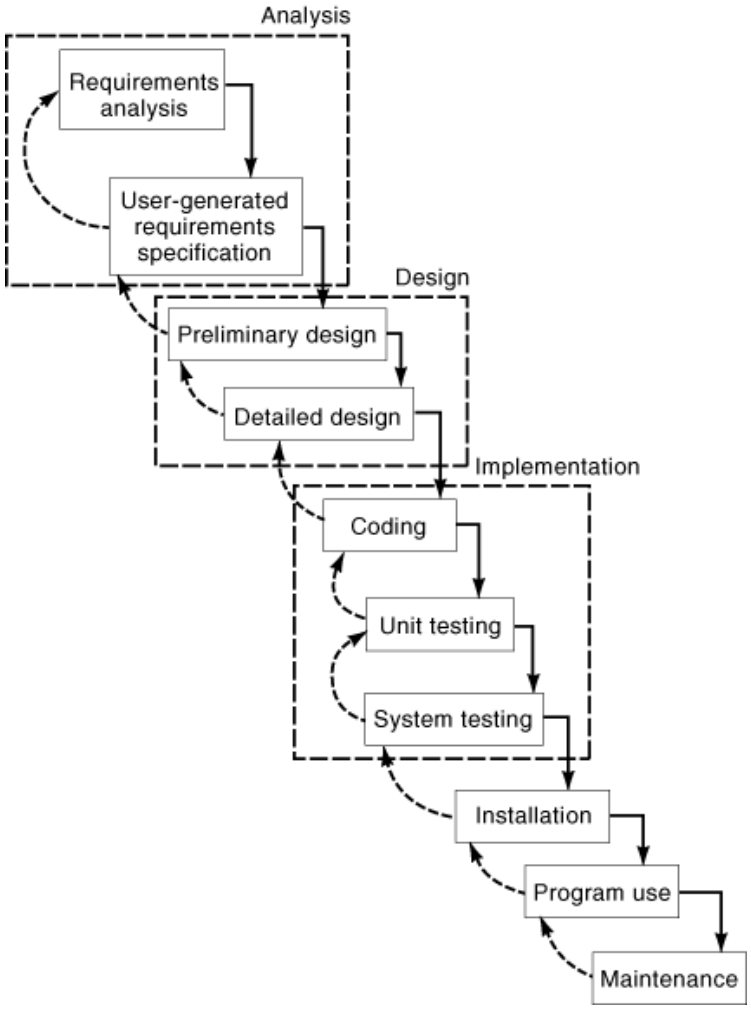

Fig. 2. The waterfall model. Every stage of the development process is closed before the next stage commences. Prototyping is sparingly encouraged.

(9) Program release
(10) Maintenance and evolution

Though structurally similar to the waterfall model, the fountain model differs in that it allows heavy prototyping. Therefore, the transition of the Fountain model from one phase to the other is something that needs to be carefully monitored for schedule slippage. The point of this exercise is to identify the right time to stop the iterative process from overrunning the allocated time and resources specific to that phase.

Another model, which attempts to integrate the conventional waterfall flow with the iterative approach adopted by prototyping, is the *object-oriented spiral model*. The spiral comprises four activities within an iteration in the life cycle of a product:

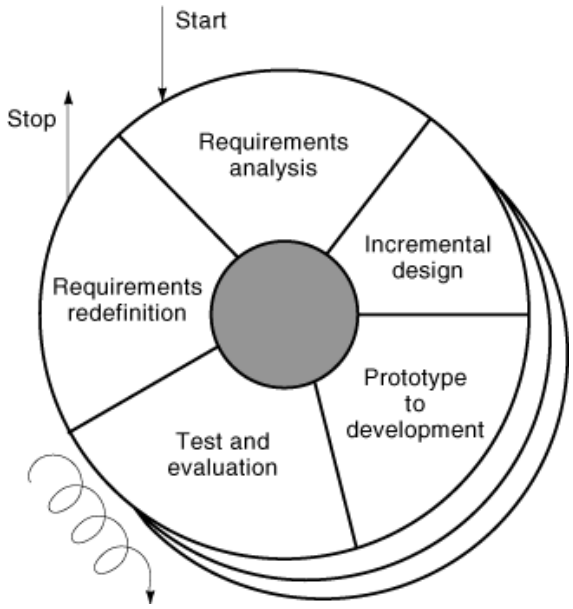

Fig. 3. Rapid prototyping. This approach offers room for design changes, making it suited for unique projects with several unknowns.

(1) Analysis
(2) Design
(3) Implementation
(4) Test and evaluation

The product gets progressively refined with each iteration of the above-mentioned cycle, as shown in Fig. 5. Iterative paradigms represented by the fountain and the spiral models are best suited for the development of large-scale system software. The choice of an *SDLC* is, however, largely governed by the nature of the work environment, the resource constraints, and the risks involved. It is important to understand that a comparative study of the benefits of one model over the other does not really serve any purpose if there exists no correlation of attributes between the processes they are being applied to.

Yet another *SDLC* model, used extensively in the defense industry, is the *clean-room* model (6). The clean-room approach is based on error prevention rather than error correction. Software modules are formally specified and mathematically proved correct; unit testing is not done under this paradigm. Errors that evaded the theoretical correctness check are caught only during integration testing.

The Qualitative Aspects of Software Management

The science of mathematical modeling is not yet sophisticated enough to capture the complexity of human interaction. These aspects cannot be represented algorithmically and present the most challenging hurdle in the race to manage. Decisions on such issues are based largely on rules of thumb or on systematic intuition, a combination of experience and intelligent speculation. The psychological aspects of management fall outside

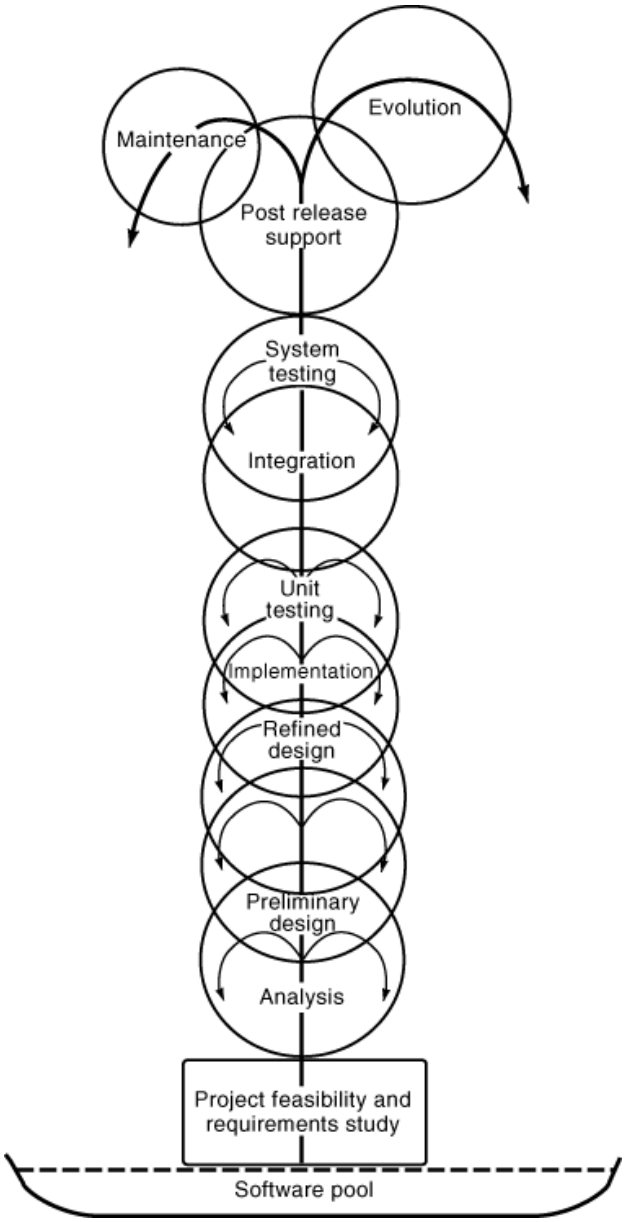

Fig. 4. The fountain model. Though similar to the waterfall model, it offers prototyping as a means of iteratively perfecting product design.

the scope of this article, wherein we restrict our discussion to the more evident and representable aspects of software management.

Managing People. Management of the work force from an engineering point of view involves getting maximum utility from the available workforce (7). Deploying a highly skilled work force is a necessity that is

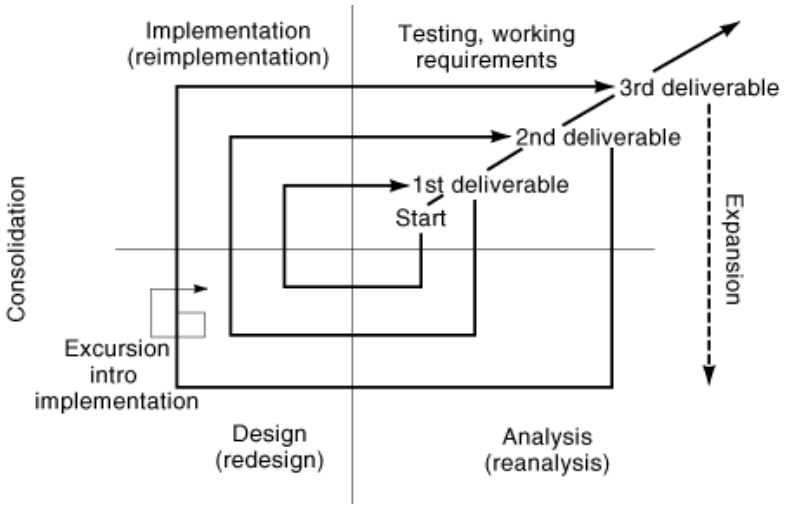

Fig. 5. The object-oriented spiral model supports prototyping across phases.

self evident. But a group of experts can prove to be ineffectual if task allocation is haphazard. A systematic approach to creating a team has its roots in the analysis done during the requirements specification stage.

Problem Decomposition. An important part of any complex project is the identification of basic blocks that can be grouped together on the basis of a common function. A typical distributed system will have entities such as the network management system, the security system, and the routing function. These functional building blocks present clean interfaces to each other, in spite of being highly complex intrinsically. For example, a network management system generally has a facility for alarm monitoring and network element configuration functionality that is intricate and highly complex. But it interacts with the other functionalities via the *simple network management protocol* (*SNMP*). All that the other entities need to know about the network management system are the basic commands offered under *SNMP*, the protocol used at the interfaces. A complex system such as this yields itself easily to problem decomposition. Specialized teams can be allocated to each of the identifiable functionalities, with a team leader identified for each group. As shown in Fig. 6 this single point of contact for each group serves to limit interactive complexity to the square of the number of team leaders involved, as opposed to having complex information flow patterns being set up between each and every member. By creating several localized views of the problem encapsulated within each team, the global complexity can be made easily manageable.

Team Definition. Team definition is fundamental to the unobstructed progress of any manufacturing process. Identifying team primes who are directly involved in the project right from its conceptual stage is an important aspect of team definition. By making the team primes responsible for the management of his or her specialized group, a controllable hierarchy of complexity is built up. Schedules are defined at each layer of this hierarchy, and the primes are held responsible for meeting localized schedules. At the next higher level, management deals with the interaction and progress of these individual groups, ensuring unhindered information flow and interaction across the interteam interfaces. Partitioning of resources between teams is made such that they do not interfere with each other's development.

Interface Definition. The complexity evident in software management arises as a result of the complexity of the artifact that is being created. The definition of encapsulated subproblems and well-defined interfaces obviates the underlying complexity considerably. To keep this complexity concealed, interfaces should be carefully defined during the requirements specification stage. Each subsystem interacts with the others through

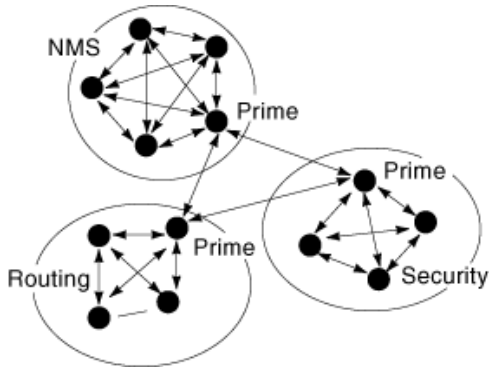

Fig. 6. A group of teams with ingteraction between primes. Team primes, acting as sources of local information, control the content and quality of information flow.

interfaces, and any changes to these interfaces can propagate to the interiors of each subsystem, affecting its internal parameters and increasing overall complexity. Thus the stability of subsystem interfaces is an important aspect of effective software management. Changes to interfaces between groups can have global ramifications that can make management of such systems a nightmare, resulting in schedule slippage and product mediocrity.

Managing Software. An important attribute of software that makes it distinct from other creative media is the ease with which it can be modified. Checking the validity and correctness of the change is, however, an esoteric science that is far from being perfected. There arise many situations where changes that were made to a section of code need to be rolled back and new changes made. Errors that show up during testing have to be corrected, and subsequent evolution has to be based on the corrected software. Thus, version control of software is an absolute necessity for the effective management of large projects involving several developers. Commercial off-the-shelf (*COTS*) software packages are available that assist in revision control and support a cooperative work environment. Such packages help in monitoring development and change as software evolves. The control of the evolution of a product with respect to change is termed *configuration management* (*CM*). A typical *CM* system maintains and records the history of the components of a product, as well as that of the whole product, in addition to coordinating concurrent changes by various team members. Most computer-aided software engineering (*CASE*) tools belong to the check-in–check-out model, having evolved from Unix development tools such as *SCCS* (Source Code Control System). Such systems offer version and access control, in addition to automated load build over individual system components. When the evolution of systems is tracked as a series of configuration versions made by various team members, the *CM* system is termed a *long transaction model*. A *CM* system that tracks logical changes is termed a change *set model*, and the *composition model* aids in the selection of the best alternative among multiple options based on predefined rules.

The Quantitative Aspects of Software Management

An essential aspect of managing software development is having a reliable way of measuring the quality of the software being developed. Faults, defects, errors, and changes are accumulated by most *CM* systems, and can be used to derive indirect product attributes. Software metrics is the study of the various approaches to quantitatively representing complex attributes of software. Some commonly used software metrics are reviewed in this section.

Another aspect of effective software management is the art of allocating resources. Cost estimation tools (8) give the software manager considerable insight into the prediction of cost and effort and help take the guesswork out of resource allocation to some extent. Having a well-defined process for a development activity gives the project more controllability, especially when things go wrong, since problem areas can be quickly identified and corrective measures put in place.

Software Metrics. There are two basic approaches to measuring the attributes of quality software. One school of thought subscribes to the axiom that good internal structure implies good external quality. The other school subscribes to the point of view that a good process implies good products. A host of product and process metrics have been developed as a result of this idea. Product metrics have been developed that measure, in their simplest form, the size of the code, and in their more sophisticated forms take complex attributes such as modularity and information flow into consideration.

Attempts have been made to quantify the functionality of software products. The initial work of DeMarco and Albrecht deserves special mention. DeMarco's research was aimed at quantifying the internal functionality of a software module. DeMarco used the term "bang," for the lack of a better word, to denote the functionality intrinsic to a software module. He defined primitives, or basic building blocks in software, such as data elements, objects, and transitions, and used them to build a composite measure.

Albrecht introduced the concept of *function points*, which were meant to capture the functionality intrinsic to a software product. The function points were attributes such as the inputs, outputs, file systems, and other input/output functionalities. The number of function points for each of these categories was found, and a weighted average was calculated as the overall complexity of the software module. The weights that were assigned to each of the function points could be scaled to reflect local variations and functionality. This approach, though simple and easily customizable, is not rigorous in terms of measurement theory. The dimensions of each of these function points are not clearly defined, and a weighted average across dimensionless terms can be misleading when used for comparisons.

Program complexity shows a high correlation with defects during testing. Metrics based on the complexity of programs can thus be used to identify sections of code that are highly error-prone. McCabe's cyclomatic complexity and object-oriented metrics (such as depth of inheritance) attempt to detect attributes like low comprehensibility and low reliability, which are strong indicators of the design being error-prone. It should be pointed out that there is always a temptation to infer cause and effect when observing a correlation. In order to assign causality authoritatively, one has to create an experiment specifically designed to provide this kind of inference. The lack of correlation is a necessary and sufficient condition to prove that two functions are not related. Conversely, the existence of correlation between two functions is necessary, but not sufficient, to prove a cause-and-effect relationship.

Ideally, the act of monitoring a process should not be instrumental in transforming the very process being measured. Models that assess the evolution of a product are usually automated and built into a configuration management system (9), to monitor the process of software development unobtrusively.

Cost Estimation. The success of a product depends not only on its quality, but also on its timely release into the market. Cost estimates are estimates of effort and elapsed time. These estimates tell us about the rate of progress of the project. Cost estimates for any project are required right from the very moment a project is conceived. An estimated delivery time for the project forms an essential part of the production "contract." Cost estimation deals with the prediction of effort level and staffing associated with developing a software artifact.

It is apparent that software products involving a large number of personnel exhibit behavior that cannot be easily extrapolated from that of a small-scale project. Software cost estimation, done during the requirements specification phase, is the very first cut at resource allocation for a project (10). Allocating too little time for a project can have disastrous consequences. On the other hand, too much time for a project can result in cost overruns as a result of the phenomenon termed *Parkinson's law*, which states that "work expands so as to fill the time available for its completion." To add to the nescience, cost and schedule estimation tools used in industry often fail to live up to expected accuracy levels. In spite of this limitation, such tools are extensively

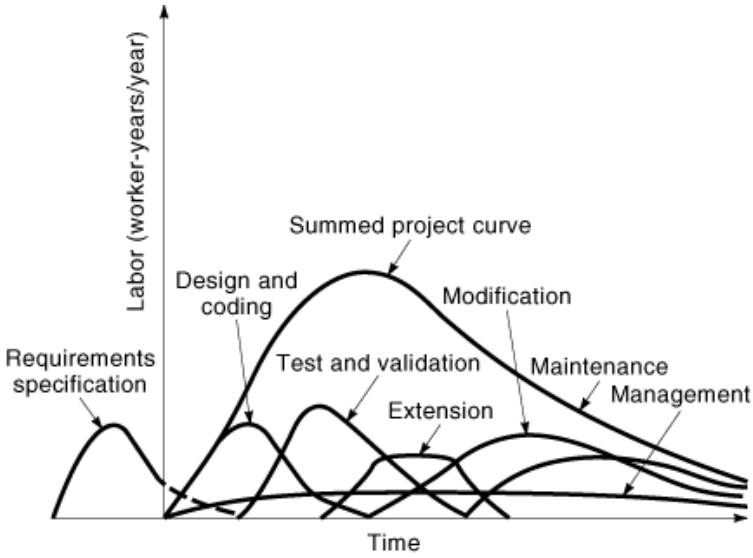

Fig. 7. Various phases of a development cycle. The staffing cycle for each phase is similar in kurtosis to the summed project curve.

used for lack of a better approach to solving the estimation problem. Models for software management and engineering are indispensable tools in controlling and managing the development of software products (11).

Intuitive solutions to problems that arise in small projects are generally not scalable to larger projects. The dynamics involved in managing a large-scale project differ considerably from those of small- or medium-scale ones. Turnaround times are longer, yet the expenses and damages that could arise as a result of delayed response can prove devastating. Prediction systems that provide timely information regarding process attributes are needed throughout the life cycle of the development process. Such systems generally accept ballpark estimates as preliminary inputs. Once a project is under way, these systems continually provide refined estimates that are much needed for coordinating the process.

An important attribute of software development, distinctly different from conventional processes, is the manner in which manpower gets deployed over time. To begin with, software development allows for a higher degree of parallel development. Several interdependent modules can be developed simultaneously using efficient design methodologies. This level of parallelism cannot usually be realized with hardware systems. In other words, software development processes can support a much faster labor-force buildup than conventional processes. The labor curves that characterize software development are thus unique in some ways. This aspect of software development renders most models used in industrial engineering unsuitable in this field.

Following an elaborate study that involved several defense projects (12), L. H. Putnam observed that the staffing curve of large-scale projects follow a series of overlapping bell-shaped curves corresponding to the different phases of a project. The different phases of the development life cycle and its corresponding staffing profiles are shown in Fig. 7.

This observation was perhaps one of the most important first steps towards managing a megaproject in a systematic manner. A number of mathematical models or frameworks have been suggested since then to represent the manner in which a project evolves with time. The choice of the representative function is sometimes governed by the peculiarities of the project being modeled and the associated environment. An overview of popular mathematical models for staffing profile prediction follows.

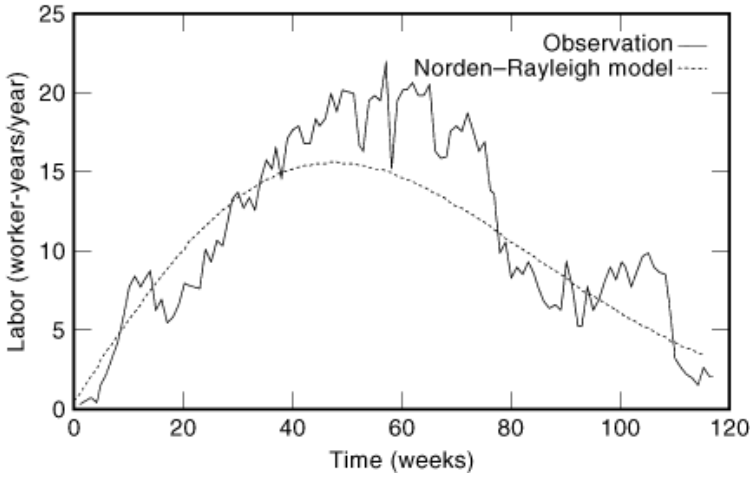

Fig. 8. The Rayleigh curve representation of the staffing profile superimposed on the actual profile.

Two models widely used in industry are the RCA PRICE S model and Putnam's *SLIM* (software life-cycle management) model. The COCOMO model (13), proposed by Boehm, provides a combination of various functional forms made accessible to the user in a structured manner. Both the Putnam and the COCOMO models use the Rayleigh distribution as an approximation to the smoothed labor distribution curve. This is based on the observation by Norden 14 that the Rayleigh distribution (Fig. 8) provides a good approximation of the manpower curve for various development processes. But Boehm points out that the slow labor-force buildup and the long tailoff time characteristic of the Rayleigh curve are not in accordance with the labor curves of most "organic-mode" projects (13). Thus, the COCOMO model uses only the central portion of the Rayleigh curve to arrive at the labor estimating equation.

An alternative model to the Rayleigh curve was proposed by F. N. Parr (15). The Parr model uses a sech^2 curve to describe a work profile. However, a well-defined methodology to estimate the parameters constituting the Parr model is not available (16). The gamma curve (17) is yet another alternative to the Rayleigh curve, when it comes to representing staffing profiles with a sharp ramp-up.

Process Improvement

One major aspect of successfully applying various metrics and models to an environment is the choice of an appropriate minimal subset of tools from the plethora of options available. It is important to note that, as in quantum mechanics, the very act of observing a certain activity can have an effect on it. Metrics collection, for this reason, should be unobtrusive, contribute to the constructive design process, and never degenerate into a tool for rating the quality of developers. Software management has overheads and in itself demands resources to be effective. Ideally, the software management team should be a highly concentrated one, absorbing minimal resources yet serving its function as the force that consolidates the project.

The Quality Improvement Paradigm. The *quality improvement paradigm* (*QIP*) is a systematic approach to improving the process of software development. This paradigm acknowledges the risks inherent in the development process. The *QIP* starts off on the assumption that every software project is unique, with an outcome that is hard to establish *a priori*. A project is seen as a dynamic state stressing entities within the development environment, such that each project flushes out weaknesses and uncovers strengths in the

process environment. The information that becomes available from a dynamic process can be iteratively used in improving the process environment. The *QIP* feedback control mechanism can be analyzed into two cycles:

(1) The control cycle involves generating feedback regarding problem identification and solution. During this cycle, the process is monitored and reevaluated to check for deviations from intended goals. The control cycle is product-specific and is geared towards delivering a specific high-quality product.
(2) The capitalization cycle involves the generation of organizational feedback. Process improvement occurs during this cycle. The metrics collected during the development process are validated *a posteriori* (18), and the effects are systematically analyzed, and process modifications implemented. Reuse and utilization of the infrastructure used for the development process are prioritized.

The Goal–Question–Metric Paradigm. The *QIP* is a long-term plan and aims at delivering a good product through an iteratively perfected process. In the context of quality improvement on a per product basis, the control cycle in itself should be finely tuned. The attributes that are being measured should be pertinent to the ultimate goal of the project. This approach is termed the *goal–question–metric* (*GQM*) paradigm and is geared towards generating a goal-based metrics program.

In the *GQM* model the analysis task should have an explicit measurement goal, and furthermore, each metric must be theoretically justifiable (19). Considerable stress is laid on meticulously documenting the rationale behind the metrics selection process. The *GQM* approach can be depicted as a layered one as shown in Fig. 9. At the highest level a goal is realized for a particular object, based on certain quality requirements and resource and environmental constraints. At the next level, the goal of the project is represented in terms of a set of questions in a quantitative manner. The point of generating a set of questions is to ensure that the complex measurement process does not lose focus in the general scheme of things. At the lowest level, the representative metrics are validated for accuracy and consistency.

Maturity Models

There is empirical evidence to support the fact that a well-defined and streamlined process is necessary to consistently create good products. It is important to be able to classify the level of sophistication a development process has attained, before it can be managed, optimized, or even compared with other processes. Considerable research has been done in this field, of which the work done at Motorola and Carnegie-Mellon University deserves special mention.

Six Sigma. The term *six sigma quality* was coined by Motorola. The term finds it origins in the symbol σ used to represent standard deviation in statistics. Assuming most natural phenomena in the real world work occur according to a Gaussian probability density function, the nature of the distribution is such that occurrence probabilities tend to cluster around a mean. The mean and the standard deviation characterize the bell-shaped normal curve. As we move further and further away from the mean, the probability density drops tremendously. If we take into consideration only the points that are less then six times the standard deviation distant from the mean, the probability density excluded is minimal, and can be used as a benchmark for comparisons. The term "six sigma" is used to imply a high level of refinement in the process, such that defects will hardly ever result. Average processes and products generally are classified at the 3σ level. The best and most sophisticated processes and products are at 6σ level. The philosophy of six sigma quality is simply quality enhancement through defect and cycle-time reduction.

The methodology of implementing six sigma quality involves six steps:

(1) Identification of product/services
(2) Identification of customers

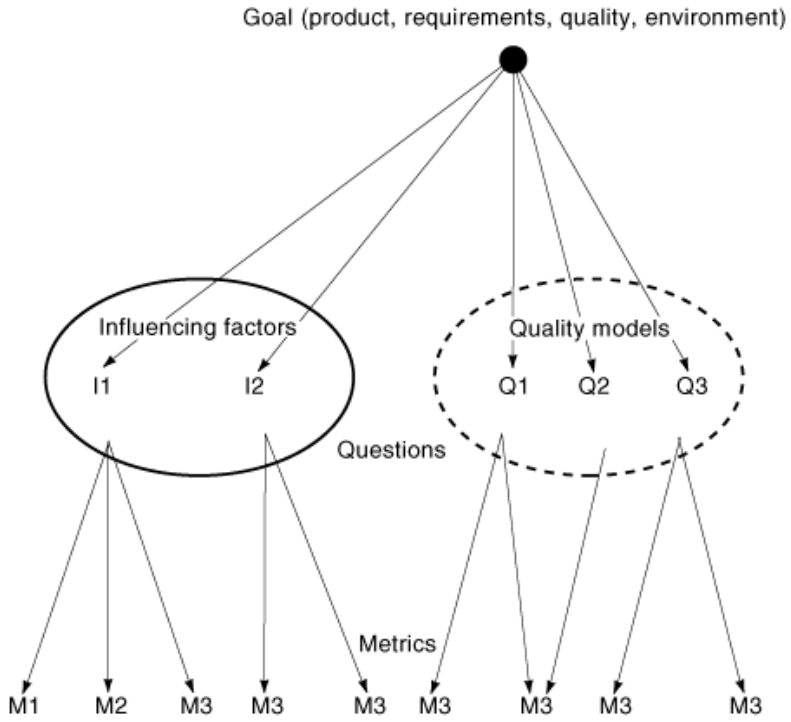

Fig. 9. The goal–question–metric paradigm ensures usefulness of metrics through traceable levels of indirection.

(3) Identification of needs
(4) Definition of the process
(5) Fortification and streamlining of the process
(6) Ensuring continuous effort

The philosophy behind the " six steps to six sigma" (*SSSS*) concept is also echoed in the systems engineering capability maturity model (SE-CMM) from the Software Engineering Institute. The SE-CMM, like the six sigma approach, attempts to classify the essential elements within a process that are prerequisites to good systems engineering.

The SE-CMM describes the stages through which a quality process progresses as it is defined, implemented, and improved. The model provides a framework for selecting process improvement strategies, from a choice of options, by determining existing capabilities of specific processes. The most critical issues are identified, so that quality and process improvement can be initiated in a specific domain, such as software engineering or systems engineering. A capability maturity model (*CMM*) is generally used as a reference model for developing and improving a mature process (20). The six-level hierarchy of development capability in increasing order of sophistication is as follows:

(1) *The Not Performed Level (Level 0).* At this level, the organization does not have a process in place. Most solutions to problems are through trial and error.

(2) *The Performed Informally Level (Level 1).* There is a rudimentary process in place for routine activities, but the system still depends on individual expertise to solve most problems. The true attributes that can improve the project have not all been systematically analyzed and captured.
(3) *The Planned and Tracked Level (Level 2).* The process is reviewed, and base practice implementation is enforced. Corrective actions are implemented as soon as any of the tracked quality indicators fall below specifications. The base practise is, however, still custom-made for a specific project.
(4) *The Well-Defined Level (Level 3).* At this level the evolving base practice is also tracked and versioned. The base practices are implemented company-wide, and the changes made to it are carefully tracked. This allows base practices to be modified to suit other development processes within the company.
(5) *The Quantitatively Controlled Level (Level 4).* At this level, a finer understanding of the problem is achieved through quantitative analysis. Through the use of appropriate metrics and extensive data collection, mathematical models are developed to predict the behavior of the project.
(6) *The Continuously Improving Level (Level 5).* This, the highest level attainable, describes a system that is continually correcting itself against the natural tendency for process quality to deteriorate. At this level the ideal management system should be stable enough to experiment with innovation.

***ISO* Standards.** Most large-scale systems require technologies and expertise from diverse fields to come together. For example, a project involving wide-area networks that include satellite links would require the cooperation of the aerospace industry with the networking industry. In such scenarios, it is very important that the nexus be as seamless as possible to achieve maximum efficiency. To enable this, standards that can be applied across diverse work environments must exist. The International Organization for Standardization (*ISO*) was founded in 1946 expressly for this purpose (21). The *ISO* is responsible for the development of a common set of manufacturing, trade, and communication standards. The *ISO* 9000 (22) series of documents provide a comprehensive list of conformance clauses that can be applied across different work environments. The *ISO* 9001 is the most comprehensive conformance model of the series and includes clauses mentioned in the other series with regards to most issues. These documents provide industries with a framework for implementing a well-defined development process and do not tie down or constrain a development environment to any convention whatsoever.

The basic *ISO* series consists of three different models: *ISO* 9001, *ISO* 9002, and *ISO* 9003. These are not three stages in an evolutionary process, but three different models based on independent standards. It is common for a company to be registered to a particular model, say *ISO* 9001, because they chose to implement their process along the lines dictated by the model for specific reasons. The idea behind this exercise is to have a well-documented, controllable process in place. Furthermore, it is stipulated in the model that the requirements specified are meant to complement any product-specific requirements already in place, and not replace them. The *ISO* 9000 models are crafted to be outside the domain of most government stipulations, and do not encroach on or attempt to replace such standards. For example, a telecom company might be *ISO* 9000 registered, but still might be guilty of violating the regulations of the Federal Communication Commission on radiations from its products. Such violations, if established, would result in the immediate abrogation of the registration.

The *ISO* standards explicitly state that "the requirements specified in this International Standard are aimed primarily at achieving customer satisfaction by preventing nonconformity, at all stages from design through to servicing." In other words, the process must provide a methodology by which the development of the product is checked for conformance at every stage of the development cycle. The guidelines stress the importance of having accountability in the quality process. Personnel should be appointed whose sole responsibility is to detect and correct nonconformity. The quality policy should be meticulously documented, and awareness of the policies among the team members should be ensured. The management must regularly review the system to accommodate changes through refinements in the policy. The *ISO* models stress the importance of having strict

control over the product design process. The input requirements must be validated, and reviewed periodically to catch inconsistencies and to provide focus to the development process. Design changes must be approved according to a certain protocol and should be carefully versioned. The *ISO* also lays down a framework of procedures for document management. The model recommendations on issues pertaining to quality, design, and documentation focus on the most essential aspects of successful development.

A project is, however, not a static phenomenon. As a product evolves, it demands resources, and changes the very work environment that is creating it. Controlling a dynamic project involves building on the basic quality improvement strategies mentioned in the *ISO* standards. Conformance to a set of standards does not necessary imply that a project will be completed on time or that the product will beat the competitor to the market. The dynamics of a project in transition needs to be studied to manage the project within the required time constraints.

Avoiding Schedule Slippage

The core challenge in managing a software project involves delivering a marketable product of optimal quality within the shortest possible time frame, consuming the least amount of resources in the process (23). Mathematical modeling, as we discussed earlier, provides a way of capturing, at least partially, the aspects of successfully managing a project. A representation of these pertinent attributes in notational or mathematical form allows one to control, optimize, and manage the development process to a considerable degree. It is a well-known fact that most cooperative development activities cannot be expedited by simply increasing the number of personnel involved in the activity. This issue was formalized within the context of software engineering by F. P. Brooks in his highly insightful collection of essays on software engineering (24). Brooks's law states that "adding manpower to a late software project makes it later." Brooks attributes this phenomenon to two factors:

- The basic nature of most development activities is sequential, at least to a certain extent. Interdependencies between subtasks often enforce a strict precedence order in their execution. The project cannot complete on schedule unless these critical subtasks can be carried out in a timely, yet sequential, manner.
- A certain degree of speedup can be achieved by increasing the labor deployed, but only to a point. Constructive communication between members of a team is fundamental to a cooperative work environment, and the more the members, the more the complexity of information flow paths.

Perhaps Brooks (24), more than others, has emphasized the need for a tool that gauges the optimal staffing level for a project in his observation that

> More software project have gone awry for lack of calendar time than for all other causes combined.

The proposition states that when management perceives the progress of a project as being behind schedule, the knee-jerk reaction is to attempt a speedup by adding extra staff. But experience has shown that this brute force approach can prove dangerous beyond a certain point. For large teams, the interaction complexity between personnel and that between management and individuals gets stressed enough to have an adverse effect on productivity. The efficacy of the developmental process is thus largely dependent on the synergy that exists between individual members of the team.

Given a staffing profile, the point on the profile beyond which overstaffing might occur can be approximated. The value of this estimate cannot be underestimated, since overstaffing is one major cause for the failure of many projects. The productivity of a team is dependent on its inherent skill level. Norden described the rate of accomplishment of a process as being proportional to two quantities:

(1) The level of skill deployable at any instant
(2) The amount of work left to be done

Based on Norden's conjectures, the skill level for a project can be derived from any generic model (25) that provides a profile for the rate of accomplishment. Ideally, the skill level would increase monotonically with time. But in reality, skill tends to saturate, and an increase in labor beyond this point can yield regenerative schedule slippage. Computing the turning values on the skill profile can give predictive knowledge about potential staff overloading.

An important aspect of controlling schedule slippage is monitoring the sequencing of tasks and identifying potential bottlenecks. Tools such as PERT (26) and CPM (27), used in operations research, can help identify the critical path. This is important, since schedule slippage is most likely to occur if subtasks along the critical path of a project gets delayed. Typically, PERT is used in conjunction with mathematical models to help track evolving projects, and to provide early warning in the event of schedule slippage.

PERT has been used in research projects since the late fifties as a reliable scheduling and controlling tool. PERT was developed for scheduling R&D activities for the US Navy. It was first used effectively in the *Polaris* missile program. PERT makes use of a probabilistic estimate of the time schedule, helps identify critical activities, and finally can be used as an iterative tool as the project matures. The network diagram shown in Fig. 10 is essentially a PERT network chart that can be iteratively refined as attributes of the project become increasingly more precise and measurable.

PERT methodology consists of three phases:

- *Planning.* During this phase, the entire project is broken down into its basic activities. The network diagram showing activity interdependencies is constructed. Network diagrams help in the analysis and design, since they provide a medium for analyzing different tasks and their interdependencies.
- *Scheduling.* This phase yields a time chart that shows the starting and finishing times of each task in the project. Once the tasks have been identified with some degree of preciseness, the time chart helps in identifying the *critical path*. Critical paths consist of tasks, that if delayed, would inevitably delay the whole project.
- *Controlling.* This phase is a highly iterative one. The network diagram can evolve as the project matures, yielding schedule estimates of increasing refinement. *Resource allocation* decisions can be made, or existing ones revisited, based on information yielded by the network diagram.

The lifetime of software, from its time of conception to its delivery, can be depicted using an activity chart as shown in Fig. 10. Conventional charts, such as the Gantt bar diagram, fail to represent the interdependency between activities. The network diagram in Fig. 10, on the other hand, captures the interdependency between activities, with each arc representing an activity, and each node representing an event or a milestone. For example the node labeled "Implementation" in Fig. 10 represents the point in time at which the implementation of the unit it refers to is complete for all practical purposes. The resulting network is a directed graph, the arrow indicating the precedence relationship of events.

Important milestones in an *SDLC* correspond to nodes (events) in a network diagram. The activity that leads from one milestone to the next is represented by a directed arc. Some milestones require the attainment of other milestones, even though they do not directly depend on them. Such dependencies are represented with dashed lines. Any level of representational granularity can be achieved by recursively applying PERT to each of the nodes themselves.

The schedule of a project is not something that can be controlled at the grass-roots level. Considerable supervision, monitoring, and control is required at the highest level to regulate and coordinate various subtasks within a project. It is only natural for developers, engrossed in the complexity of their task, to be oblivious to

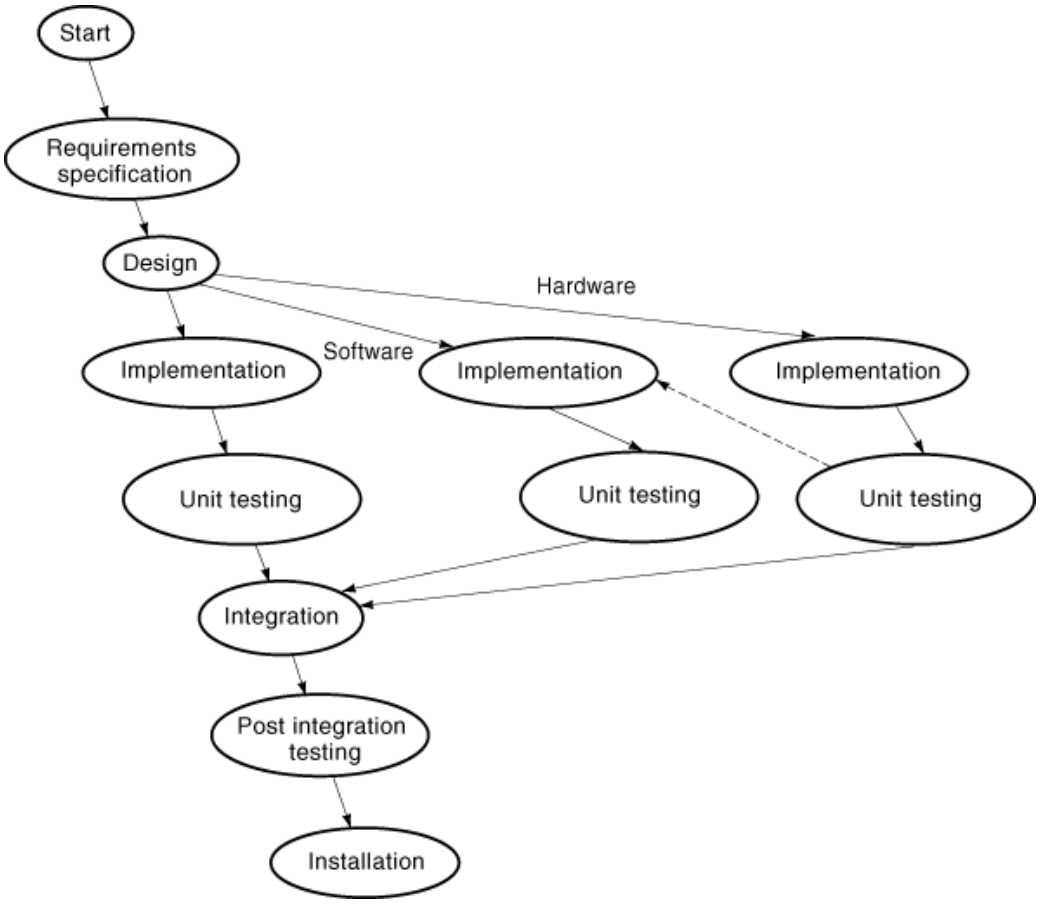

Fig. 10. A project modeled as a network. Such a representation graphically captures the sequencing of various phases and their interdependency.

the global project view. Hence the need to identify personnel who are responsible for monitoring the critical path, and flagging potential danger, is absolute (28).

Conclusions

Software management involves managing the product as well as the process of software development. Since software design is still a labor-intensive process, the role of the human psyche cannot be ignored. Effective management of a team of developers depend, to a large extent, on good leadership, and its ability to inspire the workforce to become an cohesive team dedicated to developing the project. It should be emphasized that for the various techniques mentioned in software engineering literature to be effective, the ability of management to win the trust of the team members is an essential prerequisite. Employee dissatisfaction is a major issue in today's industry; many a project has gone awry due to the untimely departure of essential team members. It is true that, in principle, projects at level 2 and above are not run by heroes and that every member is expected to be dispensable. But this hardly works in practice, as innovation requires unique skills, and the

overheads involved in training new personnel and getting them up to speed with an evolving project can prove to be severely stressful. A demonstration of genuine interest by the management in the welfare of their subordinates can go a long way in increasing productivity. Periodic assessment of the job satisfaction level among team members can help in ensuring that each team member gets an opportunity to contribute to his or her full potential and to realign his or her interests with that of the company. Some companies provide, in addition to a regular top-down annual competence review, a bottom-up approach to generate feedback for upper management. This approach is beneficial in some respects, since mistakes are more expensive as one climbs up the managerial hierarchy. Being a relatively new field, software engineering has still a lot to learn from personnel management techniques used in other areas. But being a field with abundant intellectual resources and a history of systematic thinking, there is little doubt that software management will redefine the tenets of management in a more systematic and rigorous way for the next millennium.

BIBLIOGRAPHY

1. D. J. Reifer, ed. *Tutorial: Software Management*, New York: IEEE Computer Society.
2. I. Somerville, *Software Engineering*, Reading, MA: Addison-Wesley, 1996, Ch. 29, pp. 589–610.
3. J. J. Marciniak, ed., *Encyclopedia of Software Engineering*, New York: Wiley, 1994.
4. H. Ishii, M. Kobayashi, K. Arita, Iterative design of seamless collaborative media, *Commun. ACM*, **37**(8): 83–97, 1994.
5. IEEE Computer Society, *IEEE Standard for Developing Software Life Cycle Processes*, New York: IEEE, 1996.
6. R. C. Linger, C. J. Trammell, *Cleanroom Software Engineering Reference*, Tech. rep., Pittsburgh, PA: Software Engineering Institute, Carnegie-Mellon University, 1996.
7. D. Phan, D. Vogel, J. Nunamaker, The search for perfect project management, *Computer World*, September, pp. 95–100, 1988.
8. L. H. Putnam, W. Myers, *Measures for Excellence*, Englewood Cliffs, NJ: Prentice-Hall, 1992. This book was reviewed and recommended by Ed Yourdon.
9. K. Pillai, *et al.* A configuration management system with evolutionary prototyping, *Proc. 4th Int. Symp. Appl. Corp. Comput.*, Monterrey, Mexico: ITESM, 1996.
10. L. H. Putnam, W. Myers, How solved is the cost estimation problem, *IEEE Softw.*, **14**(6): 105–107, 1997.
11. S. N. Mohanty, Software cost estimation: Present and future, *Softw. Prac. Exper.*, **11**(2), 1981.
12. L. H. Putnam, A general empirical solution to the macro software sizing and estimation problem, *IEEE Trans. Softw. Eng.*, **SE4**: 345–361, 1978.
13. B. W. Boehm, *Software Engineering Economics*, Englewood Cliffs, NJ: Prentice-Hall, 1981.
14. P. V. Norden, Curve fitting for a model of applied research and development scheduling, *IBM J. Res. Dev.*, **3**(2): 232–248, 1994.
15. F. N. Parr, An alternative to the Rayleigh curve model for software development effort, *IEEE Trans. Softw. Eng.*, **SE6**: 291–296, 1980.
16. V. R. Basili, Resource models, in *Tutorial on Models and Metrics for Software Management and Engineering*, IEEE Catalog No. EHO 167-7: New York: IEEE, pp. 4–9, 1980.
17. K. Pillai, V. S. S. Nair, A model for software development effort and cost estimation, *IEEE Trans. Softw. Eng.*, **23**: 485–497, 1997.
18. N. F. Schneidewind, Validating metrics for ensuring space shuttle flight software quality, *IEEE Computer*, **27**(8): 50–57, 1994.
19. R. E. Park, W. B. Goethert, W. A. Florac, *Goal-Driven Software Measurement: A Guidebook*, Pittsburgh, PA: Software Engineering Institute, Carnegie-Mellon University, 1996.
20. M. Paulk, *Capability Maturity Model for Software*, Tech. rep., Pittsburg, PA: Software Engineering Institute, Carnegie-Mellon University, 1987.
21. R. W. Peach, ed., *The ISO 9000 Handbook*, Fairfax, VA: CEEM Information Services, 1995.
22. M. Paulk, *A Comparison of ISO 9001 and the Capability Maturity Model for Software*, Tech. rep., Pittsburgh, PA: Software Engineering Institute, Carnegie-Mellon University, 1994.

23. M. Genutchen, Why is software late? An empirical study of reasons for delay in software development, *IEEE Trans. Softw. Eng.*, **17**: 582–590, 1991.
24. F. P. Brooks, *The Mythical Man-Month*, Reading, MA: Addison-Wesley, 1975.
25. K. Pillai, V. S. S. Nair, Early prediction of project schedule slippage, *Proc. IEEE Workshop Appl. Specific Softw. Eng. Technol.*, March 1998.
26. J. D. Wiest, F. K. Levy, *A Management Guide to PERT/CPM*, Englewood Cliffs, NJ: Prentice-Hall, 1977.
27. J. Horowitz, *Critical Path Scheduling*, Malabar, FL: Krieger Publ. Co., 1980.
28. T. K. Abdel-Hamid, S. E. Madnik, The dynamics of software project scheduling, *Commun. ACM*, **26**: 340–346, 1983.

K PILLAI
Southern Methodist University
V. S. S. NAIR
Southern Methodist University