MARUDHAR KESARI JAIN COLLEGE FOR WOMEN

SUBJECT NAME: SOFTWARE ENGINEERING.
Subject code: CSCA55

Unit-1: INTRODUCTION TO EVOLVING SOFTWARE Teaching Hours: 6 Hrs.

Evolving Role of Software – Nature of Software – Software Engineering – The Software Process– Software Engineering Practices – Software Myths – A Generic View of Process Model – Process Assessment and Improvement – Process Models : Waterfall Model – Incremental Process Models – Evolutionary Process Models – Concurrent Models.

Unit-2: REQUIREMENTS ENGINEERING Teaching Hours: 7 Hrs.

Requirements Engineering: Establishing the Groundwork – Initiating the Requirements Engineering Process – Eliciting Requirements – Collaborative Requirements Gathering – Quality Function Deployment – Usage Scenarios – Elicitation work Products – Building the Requirements Model – Elements of Requirements Model – Analysis Pattern – Requirements Analysis – Data Modeling Concepts.

Unit-3: DATA ENGINEERING Teaching Hours: 9 Hrs.

Data Engineering: Design Process and Design Quality – Design Concepts – The Design Model - Creating an Architectural Design – Software Architecture – Data Design – Architectural style – Architectural Design – Architectural Mapping Using Data Flow – Performing User Interface Design – Golden Rules.

Unit-4: TESTING STRATEGIES Teaching Hours: 10 Hrs.

Testing Strategies: Strategic Approach to Software Testing – Strategic Issues – Test Strategies for Conventional and Object Oriented Software – Validation Testing – System Testing – Art of Debugging. Software Testing Fundamentals – White Box Testing – Basis Path Testing – Control Structure Testing – Black Box Testing – Model Based Testing.

Unit-5: PROJECT MANAGEMENT Teaching Hours: 7 Hrs.

Project Management: Management Spectrum – People – Product – Process – Project – Critical Practices – Estimation: Project Planning Process – Software Scope and Feasibility – Resources – Software Project Estimation – Project Scheduling – Quality Concepts – Software Quality Assurance – Elements of Software Quality Assurance – Formal Technical Reviews.

SOFTWARE ENGINEERING

UNIT V

What is it? Although many of us (in our darker moments) take Dilbert's view of "management," it remains a very necessary activity when computer-based systems and products are built. Project management involves the planning, monitoring, and control of the people, process, and events that occur as software evolves from a preliminary concept to full operational deployment.

Who does it? Everyone "manages" to some extent, but the scope of management activities varies among people involved in a software project.
A software engineer manages her day-to-day activities, planning, monitoring, and controlling technical tasks. Project managers plan, monitor, and control the work of a team of software engineers. Senior managers coordinate the interface between the business and software professionals.

Why is it important? Building computer software is a complex undertaking, particularly if it involves many people working over a relatively long time. That's why software projects need to be managed.

What are the steps? Understand the four P's— people, product, process, and project.

People must be organized to perform software work effectively. Communication with the customer and other stakeholders must occur so that product scope and requirements are understood. A process that is appropriate for the people and the product should be selected. The project must be planned by estimating effort and calendar time to accomplish work tasks: defining work products, establishing quality checkpoints, and identifying mechanisms to monitor and control work defined by the plan.

What is the work product? A project plan is produced as management activities commence. The plan defines the process and tasks to be conducted, the people who will do the work, and the mechanisms for assessing risks, controlling change, and evaluating quality.

How do I ensure that I've done it right? You're never completely sure that the project plan is right until you've delivered a high-quality product on time and within budget. However, a project manager does it right when he encourages software people to work together as an effective team, focusing their attention on customer needs and product quality.

THE MANAGEMENT SPECTRUM

Effective software project management focuses on the four P's: people, product, process, and project. The order is not arbitrary. The manager who forgets that software engineering work is an intensely human endeavor will never have success in project management.

A manager who fails to encourage comprehensive stakeholder communication early in the evolution of a product risks building an elegant solution for the wrong problem.

The manager who pays little attention to the process runs the risk of inserting competent technical methods and tools into a vacuum. The manager who embarks without a solid project plan jeopardizes the success of the project

The People

The cultivation of motivated, highly skilled software people has been discussed since the 1960s. In fact, the "people factor" is so important that the Software Engineering Institute has developed a People Capability Maturity Model (People-CMM), in recognition of the fact that "every organization needs to continually improve its ability to attract, develop, motivate, organize, and retain the workforce needed to accomplish its strategic business objectives" .

The people capability maturity model defines the following key practice areas for software people: staffing, communication and coordination, work environment, performance management, training, compensation, competency analysis and development, career development, workgroup development, team/culture development, and others. Organizations that achieve high levels of People-CMM maturity have a higher likelihood of implementing effective software project management practices. The People-CMM is a companion to the Software Capability Maturity Model– Integration (Chapter 30) that guides organizations in the creation of a mature CHAPTER 24 PROJECT MANAGEMENT CONCEPTS 647 project . . . . . .660 software scope . . . . . .656 software team . . . . . .651 stakeholders . .649 team leaders . .650 W5 HH principle . . . . . .661 software process. Issues associated with people management and structure for software projects are considered later in this chapter. 24.1.2 The Product Before a project can be planned, product objectives and scope should be established, alternative solutions should be considered, and technical and management constraints should be identified. Without this information, it is impossible to define reasonable (and accurate) estimates of the cost, an effective assessment of risk, a realistic breakdown of project tasks, or a manageable project schedule that provides a meaningful indication of progress. As a software developer, you and other stakeholders must meet to define product objectives and scope. In many cases, this activity begins as part of the system engineering or business process engineering and continues as the first step in software requirements engineering .

Objectives identify the overall goals for the product (from the stakeholders' points of view) without considering how these goals will be achieved. Scope identifies the primary data, functions, and behaviors that characterize the product, and more important, attempts to bound these characteristics in a quantitative manner. Once the product objectives and scope are understood, alternative solutions are considered. Although very little detail is discussed, the alternatives enable managers and practitioners to select a "best" approach, given the constraints imposed by delivery deadlines, budgetary restrictions, personnel availability, technical interfaces, and myriad other factors.

The Process

A software process (Chapters 2 and 3) provides the framework from which a comprehensive plan for software development can be established. A small number of framework activities are applicable to all software projects, regardless of their size or complexity. A number of different task sets—tasks, milestones, work products, and quality assurance points—enable the framework activities to be adapted to the characteristics of the software project and the requirements of the project team. Finally, umbrella activities—such as software quality assurance, software configuration management, and measurement—overlay the process model. Umbrella activities are independent of any one framework activity and occur throughout the process.

The Project

We conduct planned and controlled software projects for one primary reason—it is the only known way to manage complexity. And yet, software teams still struggle. In a study of 250 large software projects

between 1998 and 2004, Capers Jones [Jon04] found that “about 25 were deemed successful in that they achieved their schedule, cost, and quality objectives.

PART FOUR MANAGING SOFTWARE PROJECTS

Those who adhere to the agile process philosophy (Chapter 3) argue that their process is leaner than others. That may be true, but they still have a process, and agile software engineering still requires discipline. 35 percent, while about 175 experienced major delays and overruns, or were terminated without completion.” Although the success rate for present-day software projects may have improved somewhat, our project failure rate remains much higher than it should be.1 To avoid project failure, a software project manager and the software engineers who build the product must avoid a set of common warning signs, understand the critical success factors that lead to good project management, and develop a commonsense approach for planning, monitoring, and controlling the project. Each of these issues is discussed in Section 24.5 and in the chapters that follow. 24.2 PEOPLE In a study published by the IEEE [Cur88], the engineering vice presidents of three major technology companies were asked what was the most important contributor to a successful software project. They answered in the following way: VP 1: I guess if you had to pick one thing out that is most important in our environment, I’d say it’s not the tools that we use, it’s the people. VP 2: The most important ingredient that was successful on this project was having smart people . . . very little else matters in my opinion. . . . The most important thing you do for a project is selecting the staff. . . . The success of the software development organization is very, very much associated with the ability to recruit good people. VP 3: The only rule I have in management is to ensure I have good people—real good people—and that I grow good people—and that I provide an environment in which good people can produce. Indeed, this is a compelling testimonial on the importance of people in the software engineering process. And yet, all of us, from senior engineering vice presidents to the lowliest practitioner, often take people for granted. Managers argue (as the preceding group had) that people are primary, but their actions sometimes belie their words. In this section I examine the stakeholders who participate in the software process and the manner in which they are organized to perform effective software engineering. 24.2.1

The Stakeholders

The software process (and every software project) is populated by stakeholders who can be categorized into one of five constituencies: 1. Senior managers who define the business issues that often have a significant influence on the project. 2. Project (technical) managers who must plan, motivate, organize, and control the practitioners who do software work. 3. Practitioners who deliver the technical skills that are necessary to engineer a product or application. 4. Customers who specify the requirements for the software to be engineered and other stakeholders who have a peripheral interest in the outcome. 5. End users who interact with the software once it is released for production use. Every software project is populated by people who fall within this taxonomy.2 To be effective, the project team must be organized in a way that maximizes each person’s skills and abilities. And that’s the job of the team leader. 24.2.2 Team Leaders Project management is a people-intensive activity, and for this reason, competent practitioners often make poor team leaders. They simply don’t have the right mix of people skills. And yet, as Edgemon states: “Unfortunately and all too frequently it seems, individuals just fall into a project manager role and become accidental project managers” [Edg95]. In an excellent book of technical leadership, Jerry Weinberg [Wei86] suggests an MOI model of leadership: Motivation.

The ability to encourage (by “push or pull”) technical people to produce to their best ability. Organization. The ability to mold existing processes (or invent new ones) that will enable the initial

concept to be translated into a final product. Ideas or innovation. The ability to encourage people to create and feel creative even when they must work within bounds established for a particular software product or application. Weinberg suggests that successful project leaders apply a problem-solving management style.

That is, a software project manager should concentrate on understanding the problem to be solved, managing the flow of ideas, and at the same time, letting everyone on the team know (by words and, far more important, by actions) that quality counts and that it will not be compromised.

Another view [Edg95] of the characteristics that define an effective project manager emphasizes four key traits:

Problem solving. An effective software project manager can diagnose the technical and organizational issues that are most relevant, systematically structure a solution or properly motivate other practitioners to develop the solution, apply lessons learned from past projects to new situations, and remain flexible enough to change direction if initial attempts at problem solution are fruitless. Managerial identity. A good project manager must take charge of the project. She must have the confidence to assume control when necessary and the assurance to allow good technical people to follow their instincts. Achievement. A competent manager must reward initiative and accomplishment to optimize the productivity of a project team. She must demonstrate through her own actions that controlled risk taking will not be punished. Influence and team building. An effective project manager must be able to "read" people; she must be able to understand verbal and nonverbal signals and react to the needs of the people sending these signals. The manager must remain under control in high-stress situations. 24.2.3 The Software Team There are almost as many human organizational structures for software development as there are organizations that develop software. For better or worse, organizational structure cannot be easily modified. Concern with the practical and political consequences of organizational change are not within the software project manager's scope of responsibility. However, the organization of the people directly involved in a new software project is within the project manager's purview. The "best" team structure depends on the management style of your organization, the number of people who will populate the team and their skill levels, and the overall problem difficulty. Mantei [Man81] describes seven project factors that should be considered when planning the structure of software engineering teams: • Difficulty of the problem to be solved • "Size" of the resultant program(s) in lines of code or function points • Time that the team will stay together (team lifetime) • Degree to which the problem can be modularized • Required quality and reliability of the system to be built • Rigidity of the delivery date • Degree of sociability (communication) required for the project Constantine [Con93] suggests four "organizational paradigms" for software engineering teams:

1. A closed paradigm structures a team along a traditional hierarchy of authority. Such teams can work well when producing software that is quite similar to past efforts, but they will be less likely to be innovative when working within the closed paradigm.
2. A random paradigm structures a team loosely and depends on individual initiative of the team members. When innovation or technological breakthrough is required, teams following the random paradigm will excel. But such teams may struggle when "orderly performance" is required. 3. An open paradigm attempts to structure a team in a manner that achieves some of the controls associated with the closed paradigm but also much of the innovation that occurs when using the random paradigm. Work is performed collaboratively, with heavy communication and consensus-based decision making the trademarks of open paradigm teams. Open paradigm team structures are well suited to the solution

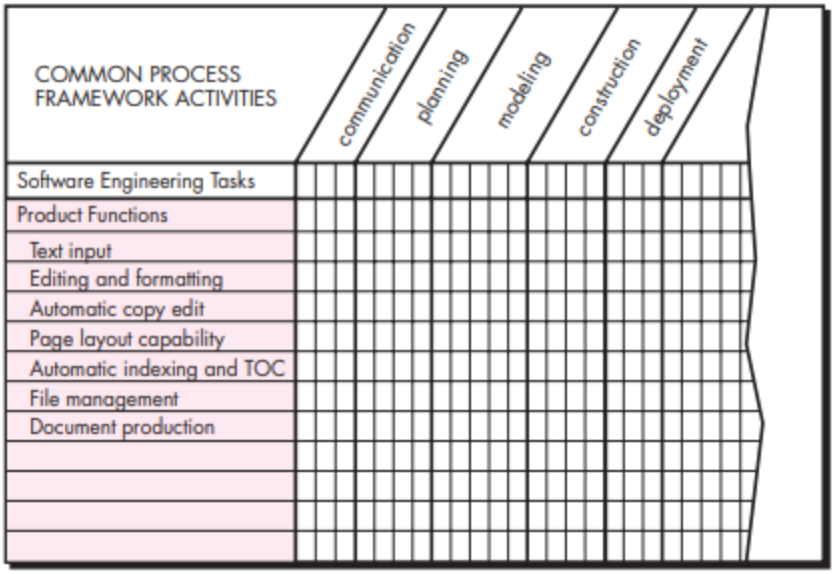
of complex problems but may not perform as efficiently as other teams. 4. A synchronous paradigm relies on the natural compartmentalization of a problem and organizes team members to work on pieces of the problem with little active communication among themselves. As an historical footnote, one of the earliest software team organizations was a closed paradigm structure originally called the chief programmer team. This structure was first proposed by Harlan Mills and described by Baker [Bak72]. The nucleus of the team was composed of a senior engineer (the chief programmer), who plans, coordinates, and reviews all technical activities of the team; technical staff (normally two to five people), who conduct analysis and development activities; and a backup engineer, who supports the senior engineer in his or her activities and can replace the senior engineer with minimum loss in project continuity. The chief programmer may be served by one or more specialists (e.g., telecommunications expert, database designer), support staff (e.g., technical writers, clerical personnel), and a software librarian. As a counterpoint to the chief programmer team structure, Constantine's random paradigm [Con93] suggests a software team with creative independence whose approach to work might best be termed innovative anarchy. Although the free-spirited approach to software work has appeal, channeling creative energy into a highperformance team must be a central goal of a software engineering organization. To achieve a high-performance team: • Team members must have trust in one another• The distribution of skills must be appropriate to the problem. • Mavericks may have to be excluded from the team, if team cohesiveness is to be maintained. Regardless of team organization, the objective for every project manager is to help create a team that exhibits cohesiveness. In their book, Peopleware, DeMarco and Lister [DeM98] discuss this issue:
: (1) a frenzied work atmosphere, (2) high frustration that causes friction among team members, (3) a "fragmented or poorly coordinated" software process, (4) an unclear definition of roles on the software team, and (5) "continuous and repeated exposure to failure." To avoid a frenzied work environment, the project manager should be certain that the team has access to all information required to do the job and that major goals and objectives, once defined, should not be modified unless absolutely necessary. A software team can avoid frustration if it is given as much responsibility for decision making as possible. An inappropriate process (e.g., unnecessary or burdensome work tasks or poorly chosen work products) can be avoided by understanding the product to be built, the people doing the work, and by allowing the team to select the process model. The team itself should establish its own mechanisms for accountability (technical reviews3 are an excellent way to accomplish this) and define a series of corrective approaches when a member of the team fails to perform. And finally, the key to avoiding an atmosphere of failure is to establish team-based techniques for feedback and problem solving. In addition to the five toxins described by Jackman, a software team often struggles with the differing human traits of its members. Some team members are extroverts; others are introverts. Some people gather information intuitively, distilling broad concepts from disparate facts. Others process information linearly, collecting and organizing minute details from the data provided. Some team members are comfortable making decisions only when a logical, orderly argument is presented. Others are intuitive, willing to make a decision based on "feel." Some practitioners want a detailed schedule populated by organized tasks that enable them to achieve closure for some element of a project. Others prefer a more spontaneous environment in which open issues are okay. Some work hard to get things done long before a milestone date, thereby avoiding stress as the date approaches, while others are energized by the rush to make a last-minute deadline. A detailed discussion of the psychology of these traits and the ways in which a skilled team leader can help people with opposing traits to work together is beyond the scope of this book.4 However, it is important to note that recognition of human differences is the first step toward creating teams that jell. 24.2.4 Agile Teams Over the past decade, agile software development (Chapter 3) has been suggested as an antidote to many of the problems that have plagued software project work. To review, the agile philosophy encourages customer satisfaction and early incremental delivery of software, small highly motivated project teams, informal methods, minimal

software engineering work products, and overall development simplicity. The small, highly motivated project team, also called an agile team, adopts many of the characteristics of successful software project teams discussed in the preceding section and avoids many of the toxins that create problems. However, the agile philosophy stresses individual (team member) competency coupled with group collaboration as critical success factors for the team. Cockburn and Highsmith [Coc01a] note this when they write: If the people on the project are good enough, they can use almost any process and accomplish their assignment. If they are not good enough, no process will repair their inadequacy—"people trump process" is one way to say this. However, lack of user and executive support can kill a project—"politics trump people." Inadequate support can keep even good people from accomplishing the job. To make effective use of the competencies of each team member and to foster effective collaboration through a software project, agile teams are self-organizing. A self-organizing team does not necessarily maintain a single team structure but instead uses elements of Constantine's random, open, and synchronous paradigms discussed in Section 24.2.3. Many agile process models (e.g., Scrum) give the agile team significant autonomy to make the project management and technical decisions required to get the job done. Planning is kept to a minimum, and the team is allowed to select its own approach (e.g., process, methods, tools), constrained only by business requirements and organizational standards. As the project proceeds, the team self-organizes to focus individual competency in a way that is most beneficial to the project at a given point in time. To accomplish this, an agile team might conduct daily team meetings to coordinate and synchronize the work that must be accomplished for that day. Based on information obtained during these meetings, the team adapts its approach in a way that accomplishes an increment of work. As each day passes, continual selforganizationandcollaborationmovetheteamtowardacompletedsoftwareincrement. 24.2.5 Coordination and Communication Issues There are many reasons that software projects get into trouble. The scale of many development efforts is large, leading to complexity, confusion, and significant difficulties in coordinating team members. Uncertainty is common, resulting in a continuing stream of changes that ratchets the project team. Interoperability has become a key characteristic of many systems. New software must communicate with existing software and conform to predefined constraints imposed by the system or product. These characteristics of modern software—scale, uncertainty, and interoperability— are facts of life. To deal with them effectively, you must establish effective methods for coordinating the people who do the work. To accomplish this, mechanisms for formal and informal communication among team members and between multiple teams must be established. Formal communication is accomplished through "writing, structured meetings, and other relatively non-interactive and impersonal communication channels" [Kra95]. Informal communication is more personal. Members of a software team share ideas on an ad hoc basis, ask for help as problems arise, and interact with one another on a daily basis

THE PRODUCT

A software project manager is confronted with a dilemma at the very beginning of a software project. Quantitative estimates and an organized plan are required, but solid information is unavailable. A detailed analysis of software requirements would provide necessary information for estimates, but analysis often takes weeks or even months to complete. Worse, requirements may be fluid, changing regularly as the project proceeds. Yet, a plan is needed "now!" Like it or not, you must examine the product and the problem it is intended to solve at the very beginning of the project. At a minimum, the scope of the product must be established and bounded. 24.3.1 Software Scope The first software project management activity is the determination of software scope. Scope is defined by answering the following questions: Context. How does the software to be built fit into a larger system, product, or business context, and what constraints are imposed as a result of the context? Information objectives. What customer-visible data objects are produced as output from the software? What data objects are required for input? Function and performance. What function does the software perform to transform

input data into output? Are any special performance characteristics to be addressed? Software project scope must be unambiguous and understandable at the management and technical levels. A statement of software scope must be bounded. That is, quantitative data (e.g., number of simultaneous users, target environment, maximum allowable response time) are stated explicitly, constraints and/or limitations (e.g., product cost restricts memory size) are noted, and mitigating factors (e.g., desired algorithms are well understood and available in Java) are described. 24.3.2 Problem Decomposition Problem decomposition, sometimes called partitioning or problem elaboration, is an activity that sits at the core of software requirements analysis (Chapters 6 and 7). During the scoping activity no attempt is made to fully decompose the problem. Rather, decomposition is applied in two major areas: (1) the functionality and content (information) that must be delivered and (2) the process that will be used to deliver it. Human beings tend to apply a divide-and-conquer strategy when they are confronted with a complex problem. Stated simply, a complex problem is partitioned into smaller problems that are more manageable. This is the strategy that applies as project planning begins. Software functions, described in the statement of scope, are evaluated and refined to provide more detail prior to the beginning of estimation (Chapter 26). Because both cost and schedule estimates are functionally oriented, 656 PART FOUR MANAGING SOFTWARE PROJECTS If you can't bound a characteristic of the software you intend to build, list the characteristic as a project risk (Chapter 25). In order to develop a reasonable project plan, you must decompose the problem. This can be accomplished using a list of functions or with use cases. some degree of decomposition is often useful. Similarly, major content or data objects are decomposed into their constituent parts, providing a reasonable understanding of the information to be produced by the software. As an example, consider a project that will build a new word-processing product. Among the unique features of the product are continuous voice as well as virtual keyboard input via a multitouch screen, extremely sophisticated "automatic copy edit" features, page layout capability, automatic indexing and table of contents, and others. The project manager must first establish a statement of scope that bounds these features (as well as other more mundane functions such as editing, file management, and document production). For example, will continuous voice input require that the product be "trained" by the user? Specifically, what capabilities will the copy edit feature provide? Just how sophisticated will the page layout capability be and will it encompass the capabilities implied by a multitouch screen? As the statement of scope evolves, a first level of partitioning naturally occurs. The project team learns that the marketing department has talked with potential customers and found that the following functions should be part of automatic copy editing: (1) spell checking, (2) sentence grammar checking, (3) reference checking for large documents (e.g., Is a reference to a bibliography entry found in the list of entries in the bibliography?), (4) the implementation of a style sheet feature that imposed consistency across a document, and (5) section and chapter reference validation for large documents. Each of these features represents a subfunction to be implemented in software. Each can be further refined if the decomposition will make planning easier. The framework activities (Chapter 2) that characterize the software process are applicable to all software projects. The problem is to select the process model that is appropriate for the software to be engineered by your project team. Your team must decide which process model is most appropriate for (1) the customers who have requested the product and the people who will do the work, (2) the characteristics of the product itself, and (3) the project environment in which the software team works. When a process model has been selected, the team then defines a preliminary project plan based on the set of process framework activities. Once the preliminary plan is established, process decomposition begins. That is, a complete plan, reflecting the work tasks required to populate the framework activities must be created. We explore these activities briefly in the sections that follow and present a more detailed view in Chapter 26. 24.4.1 Melding the Product and the Process Project planning begins with the melding of the product and the process. Each function to be engineered by your team must pass through the set of framework activities that have been defined for your software organization.



Assume that the organization has adopted the generic framework activities— communication, planning, modeling, construction, and deployment— discussed in Chapter 2. The team members who work on a product function will apply each of the framework activities to it. In essence, a matrix similar to the one shown in Figure 24.1 is created. Each major product function (the figure notes functions for the word-processing software discussed earlier) is listed in the left-hand column. Framework activities are listed in the top row. Software engineering work tasks (for each framework activity) would be entered in the following row.5 The job of the project manager (and other team members) is to estimate resource requirements for each matrix cell, start and end dates for the tasks associated with each cell, and work products to be produced as a consequence of each task. These activities are considered in Chapter 26. 24.4.2 Process Decomposition A software team should have a significant degree of flexibility in choosing the software process model that is best for the project and the software engineering tasks that populate the process model once it is chosen. A relatively small project that is similar to past efforts might be best accomplished using the linear sequential approach. If the deadline is so tight that full functionality cannot reasonably be delivered, an incremental strategy might be best. Similarly, projects with other characteristics (e.g., uncertain requirements, breakthrough technology, difficult customers, significant reuse potential) will lead to the selection of other process models. Once the process model has been chosen, the process framework is adapted to it. In every case, the generic process framework discussed earlier can be used. It will work for linear models, for iterative and incremental models, for evolutionary models, and even for concurrent or component assembly models. The process framework is invariant and serves as the basis for all work performed by a software organization. But actual work tasks do vary. Process decomposition commences when the project manager asks, “How do we accomplish this framework activity?” For example, a small, relatively simple project might require the following work tasks for the communication activity: 1. Develop list of clarification issues. 2. Meet with stakeholders to address clarification issues. 3. Jointly develop a statement of scope. 4. Review the statement of scope with all concerned. 5. Modify the statement of scope as required. These events might occur over a period of less than 48 hours. They represent a process decomposition that is appropriate for the small, relatively simple project. Now, consider a more complex project, which has a broader scope and more significant business impact. Such a project might require the following work tasks for the communication: 1. Review the customer request. 2. Plan and schedule a formal, facilitated meeting with all stakeholders. 3. Conduct research to specify the proposed solution and existing approaches. 4. Prepare a “working document” and an agenda for the formal meeting. 5. Conduct the meeting. 6. Jointly

develop mini-specs that reflect data, functional, and behavioral features of the software. Alternatively, develop use cases that describe the software from the user's point of view. 7. Review each mini-spec or use case for correctness, consistency, and lack of ambiguity. 8. Assemble the mini-specs into a scoping document. 9. Review the scoping document or collection of use cases with all concerned. 10. Modify the scoping document or use cases as required. Both projects perform the framework activity that we call communication, but the first project team performs half as many software engineering work tasks as the second.

THE PROJECT

In order to manage a successful software project, you have to understand what can go wrong so that problems can be avoided. In an excellent paper on software projects, John Reel [Ree99] defines 10 signs that indicate that an information systems project is in jeopardy: 1. Software people don't understand their customer's needs. 2. The product scope is poorly defined. 3. Changes are managed poorly. 4. The chosen technology changes. 5. Business needs change [or are ill defined]. 6. Deadlines are unrealistic. 7. Users are resistant. 8. Sponsorship is lost [or was never properly obtained]. 9. The project team lacks people with appropriate skills. 10. Managers [and practitioners] avoid best practices and lessons learned. Jaded industry professionals often refer to the 90–90 rule when discussing particularly difficult software projects: The first 90 percent of a system absorbs 90 percent of the allotted effort and time. The last 10 percent takes another 90 percent of the allotted effort and time [Zah94]. The seeds that lead to the 90–90 rule are contained in the signs noted in the preceding list. But enough negativity! How does a manager act to avoid the problems just noted? Reel [Ree99] suggests a five-part commonsense approach to software projects: 1. Start on the right foot. This is accomplished by working hard (very hard) to understand the problem that is to be solved and then setting realistic objectives and expectations for everyone who will be involved in the project. It is reinforced by building the right team (Section 24.2.3) and giving the team the autonomy, authority, and technology needed to do the job. 2. Maintain momentum. Many projects get off to a good start and then slowly disintegrate. To maintain momentum, the project manager must provide incentives to keep turnover of personnel to an absolute minimum, the team should emphasize quality in every task it performs, and senior management should do everything possible to stay out of the team's way.7 660 PART FOUR MANAGING SOFTWARE PROJECTS What are the signs that a software project is in jeopardy? ? uote: "We don't have time to stop for gas, we're already late." M. Cleron 7 The implication of this statement is that bureaucracy is reduced to a minimum, extraneous meetings are eliminated, and dogmatic adherence to process and project rules is deemphasized. The team should be self-organizing and autonomous. 3. Track progress. For a software project, progress is tracked as work products (e.g., models, source code, sets of test cases) are produced and approved (using technical reviews) as part of a quality assurance activity. In addition, software process and project measures (Chapter 25) can be collected and used to assess progress against averages developed for the software development organization. 4. Make smart decisions. In essence, the decisions of the project manager and the software team should be to "keep it simple." Whenever possible, decide to use commercial off-the-shelf software or existing software components or patterns, decide to avoid custom interfaces when standard approaches are available, decide to identify and then avoid obvious risks, and decide to allocate more time than you think is needed to complex or risky tasks (you'll need every minute). 5. Conduct a postmortem analysis. Establish a consistent mechanism for extracting lessons learned for each project. Evaluate the planned and actual schedules, collect and analyze software project metrics, get feedback from team members and customers, and record findings in written form.

CRITICAL PRACTICES

The Airlie Council8 has developed a list of "critical software practices for performance-based management." These practices are "consistently used by, and considered critical by, highly successful software projects and organizations whose 'bottom line' performance is consistently much better than industry averages" [Air99]. Critical practices9 include: metric-based project management (Chapter 25), empirical cost and schedule estimation (Chapters 26 and 27), earned value tracking (Chapter 27), defect tracking against quality targets (Chapters 14 though 16), and people aware management (Section 24.2). Each of these critical practices is addressed throughout Parts 3 and 4 of this book.

THE PRODUCT PLANINNG PROCESS

The objective of software project planning is to provide a framework that enables the manager to make reasonable estimates of resources, cost, and schedule. In addition, estimates should attempt to define best-case and worst-case scenarios so that project outcomes can be bounded. Although there is an inherent degree of uncertainty, the software team embarks on a plan that has been established as a consequence of these tasks. Therefore, the plan must be adapted and updated as the project proceeds. In the following sections, each of the actions associated with software project planning is discussed.

SOFTWARE SCOPE AND FEASABILITY

Software scope describes the functions and features that are to be delivered to end users; the data that are input and output; the "content" that is presented to users as a consequence of using the software; and the performance, constraints, interfaces, and reliability that bound the system. Scope is defined using one of two techniques: 1. A narrative description of software scope is developed after communication with all stakeholders. 2. A set of use cases3 is developed by end users. Functions described in the statement of scope (or within the use cases) are evaluated and in some cases refined to provide more detail prior to the beginning of estimation. Because both cost and schedule estimates are functionally oriented, some degree of decomposition is often useful. Performance considerations encompass processing and response time requirements. Constraints identify limits placed on the software by external hardware, available memory, or other existing systems. Once scope has been identified (with the concurrence of the customer), it is reasonable to ask: "Can we build software to meet this scope? Is the project feasible?" All too often, software engineers rush past these questions (or are pushed past them by impatient managers or other stakeholders), only to become mired in a project that is doomed from the onset. Putnam and Myers [Put97a] address this issue when they write [N]ot everything imaginable is feasible, not even in software, evanescent as it may appear to outsiders. On the contrary, software feasibility has four solid dimensions: Technology— Is a project technically feasible? Is it within the state of the art? Can defects be reduced to a level matching the application's needs? Finance—Is it financially feasible? Can development be completed at a cost the software organization, its client, or the market can afford? Time—Will the project's time-to-market beat the competition? Resources—Does the organization have the resources needed to succeed? Putnam and Myers correctly suggest that scoping is not enough. Once scope is understood, you must work to determine if it can be done within the dimensions just noted. This is a crucial, although often overlooked, part of the estimation process.

RESOURCES

The second planning task is estimation of the resources required to accomplish the software development effort. Figure 26.1 depicts the three major categories of software engineering resources—people, reusable software components, and the development environment (hardware and software

tools). Each resource is specified with four characteristics: description of the resource, a statement of availability, time when the resource will be required, and duration of time that the resource will be applied. The last two characteristics can be viewed as a time window. Availability of the resource for a specified window must be established at the earliest practical time. 26.4.1 Human Resources The planner begins by evaluating software scope and selecting the skills required to complete development. Both organizational position (e.g., manager, senior software engineer) and specialty (e.g., telecommunications, database, client-server) are

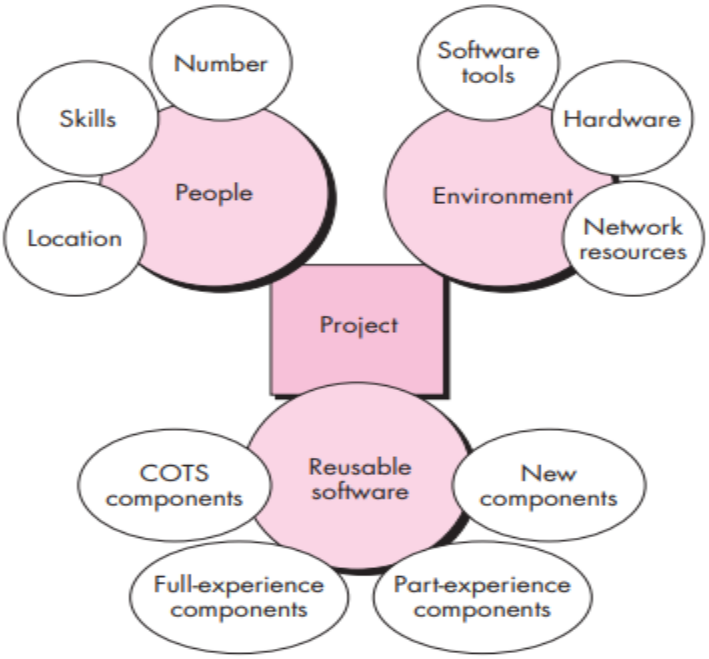


FIG: Project resources

specified. For relatively small projects (a few person-months), a single individual may perform all software engineering tasks, consulting with specialists as required. For larger projects, the software team may be geographically dispersed across a number of different locations. Hence, the location of each human resource is specified. The number of people required for a software project can be determined only after an estimate of development effort (e.g., person-months) is made. Techniques for estimating effort are discussed later in this chapter. 26.4.2 Reusable Software Resources Component-based software engineering (CBSE)4 emphasizes reusability—that is, the creation and reuse of software building blocks. Such building blocks, often called components, must be cataloged for easy reference, standardized for easy application, and validated for easy integration. Bennatan [Ben00] suggests four software resource categories that should be considered as planning proceeds: Off-the-shelf components. Existing software that can be acquired from a third party or from a past project. COTS (commercial off-the-shelf) components are purchased from a third party, are ready for use on the current project, and have been fully validated. Full-experience components. Existing specifications, designs, code, or test data developed for past projects that are similar to the software to be built for the current project. Members of the current software team have had full experience in the application area represented by these components. Therefore, modifications required for full-experience components will be relatively low risk. Partial-experience components. Existing specifications, designs, code, or test

data developed for past projects that are related to the software to be built for the current project but will require substantial modification. Members of the current software team have only limited experience in the application area represented by these components. Therefore, modifications required for partial-experience components have a fair degree of risk. New components. Software components must be built by the software team specifically for the needs of the current project. Ironically, reusable software components are often neglected during planning, only to become a paramount concern later in the software process. It is better to specify software resource requirements early. In this way technical evaluation of the alternatives can be conducted and timely acquisition can occur. 26.4.3 Environmental Resources The environment that supports a software project, often called the software engineering environment (SEE), incorporates hardware and software. Hardware provides a platform that supports the tools (software) required to produce the work products that are an outcome of good software engineering practice.5 Because most software organizations have multiple constituencies that require access to the SEE, you must prescribe the time window required for hardware and software and verify that these resources will be available. When a computer-based system (incorporating specialized hardware and software) is to be engineered, the software team may require access to hardware elements being developed by other engineering teams. For example, software for a robotic device used within a manufacturing cell may require a specific robot (e.g., a robotic welder) as part of the validation test step; a software project for advanced page layout may need a high-speed digital printing system at some point during development. Each hardware element must be specified as part of planning.

SOFTWARE PROJECT ESTIMATION

Software cost and effort estimation will never be an exact science. Too many variables—human, technical, environmental, political—can affect the ultimate cost of software and effort applied to develop it. However, software project estimation can be transformed from a black art to a series of systematic steps that provide estimates with acceptable risk. To achieve reliable cost and effort estimates, a number of options arise: 1. Delay estimation until late in the project (obviously, we can achieve 100 percent accurate estimates after the project is complete!). 2. Base estimates on similar projects that have already been completed. 3. Use relatively simple decomposition techniques to generate project cost and effort estimates. 4. Use one or more empirical models for software cost and effort estimation. Unfortunately, the first option, however attractive, is not practical. Cost estimates must be provided up-front. However, you should recognize that the longer you wait, the more you know, and the more you know, the less likely you are to make serious errors in your estimates. The second option can work reasonably well, if the current project is quite similar to past efforts and other project influences (e.g., the customer, business conditions, the software engineering environment, deadlines) are roughly equivalent. Unfortunately, past experience has not always been a good indicator of future results. The remaining options are viable approaches to software project estimation. Ideally, the techniques noted for each option should be applied in tandem; each used as a cross-check for the other. Decomposition techniques take a divide-and-conquer CHAPTER 26 ESTIMATION FOR SOFTWARE PROJECTS 697 5 Other hardware—the target environment—is the computer on which the software will execute when it has been released to the end user. uote: “In an age of outsourcing and increased competition, the ability to estimate more accurately . . . has emerged as a critical success factor for

many IT groups." Rob Thomsett approach to software project estimation. By decomposing a project into major functions and related software engineering activities, cost and effort estimation can be performed in a stepwise fashion. Empirical estimation models can be used to complement decomposition techniques and offer a potentially valuable estimation approach in their own right. A model is based on experience (historical data) and takes the form d f (vi) where d is one of a number of estimated values (e.g., effort, cost, project duration) and vi are selected independent parameters (e.g., estimated LOC or FP). Automated estimation tools implement one or more decomposition techniques or empirical models and provide an attractive option for estimating. In such systems, the characteristics of the development organization (e.g., experience, environment) and the software to be developed are described. Cost and effort estimates are derived from these data. Each of the viable software cost estimation options is only as good as the historical data used to seed the estimate. If no historical data exist, costing rests on a very shaky foundation. In Chapter 25, we examined the characteristics of some of the software metrics that provide the basis for historical estimation data.

PROJECT SCHEDULING

Fred Brooks was once asked how software projects fall behind schedule. His response was as simple as it was profound: "One day at a time." The reality of a technical project (whether it involves building a hydroelectric plant or developing an operating system) is that hundreds of small tasks must occur to accomplish a larger goal. Some of these tasks lie outside the mainstream and may be completed without worry about impact on project completion date. Other tasks lie on the "critical path." If these "critical" tasks fall behind schedule, the completion date of the entire project is put into jeopardy. As a project manager, your objective is to define all project tasks, build a network that depicts their interdependencies, identify the tasks that are critical within the network, and then track their progress to ensure that delay is recognized "one day at a time." To accomplish this, you must have a schedule that has been defined at a degree of resolution that allows progress to be monitored and the project to be controlled. Software project scheduling is an action that distributes estimated effort across the planned project duration by allocating the effort to specific software engineering tasks. It is important to note, however, that the schedule evolves over time. During early stages of project planning, a macroscopic schedule is developed. This type of schedule identifies all major process framework activities and the product functions to which they are applied. As the project gets under way, each entry on the macroscopic schedule is refined into a detailed schedule. Here, specific software actions and tasks (required to accomplish an activity) are identified and scheduled. Scheduling for software engineering projects can be viewed from two rather different perspectives. In the first, an end date for release of a computer-based system has already (and irrevocably) been established. The software organization is constrained to distribute effort within the prescribed time frame. The second view of software scheduling assumes that rough chronological bounds have been discussed but that the end date is set by the software engineering organization. Effort is distributed to make best use of resources, and an end date is defined after careful analysis of the software. Unfortunately, the first situation is encountered far more frequently than the second. 724 PART FOUR MANAGING SOFTWARE PROJECTS The tasks required to achieve a project manager's objective should not be performed manually. There are many excellent scheduling tools. Use them. uote: "Overly optimistic scheduling doesn't result in

shorter actual schedules, it results in longer ones." Steve McConnell 27.2.1 Basic Principles Like all other areas of software engineering, a number of basic principles guide software project scheduling: Compartmentalization. The project must be compartmentalized into a number of manageable activities and tasks. To accomplish compartmentalization, both the product and the process are refined. Interdependency. The interdependency of each compartmentalized activity or task must be determined. Some tasks must occur in sequence, while others can occur in parallel. Some activities cannot commence until the work product produced by another is available. Other activities can occur independently. Time allocation. Each task to be scheduled must be allocated some number of work units (e.g., person-days of effort). In addition, each task must be assigned a start date and a completion date that are a function of the interdependencies and whether work will be conducted on a full-time or part-time basis. Effort validation. Every project has a defined number of people on the software team. As time allocation occurs, you must ensure that no more than the allocated number of people has been scheduled at any given time. For example, consider a project that has three assigned software engineers (e.g., three person-days are available per day of assigned effort4). On a given day, seven concurrent tasks must be accomplished. Each task requires 0.50 person-days of effort. More effort has been allocated than there are people to do the work. Defined responsibilities. Every task that is scheduled should be assigned to a specific team member. Defined outcomes. Every task that is scheduled should have a defined outcome. For software projects, the outcome is normally a work product (e.g., the design of a component) or a part of a work product. Work products are often combined in deliverables. Defined milestones. Every task or group of tasks should be associated with a project milestone. A milestone is accomplished when one or more work products has been reviewed for quality (Chapter 15) and has been approved. Each of these principles is applied as the project schedule evolves. 27.2.2 The Relationship Between People and Effort In a small software development project a single person can analyze requirements, perform design, generate code, and conduct tests. As the size of a project increases, more people must become involved. (We can rarely afford the luxury of approaching a 10 person-year effort with one person working for 10 years!)

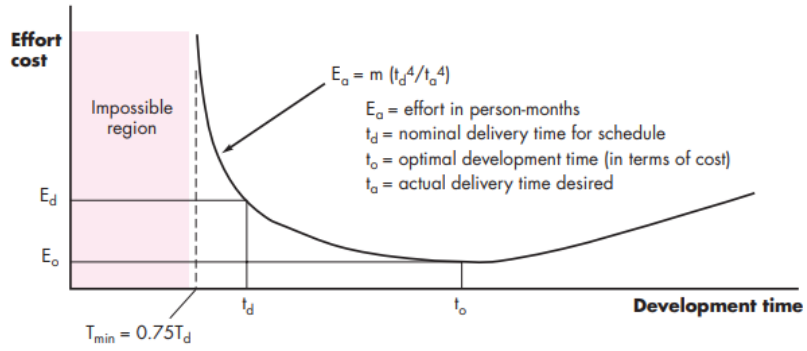


FIG: The relationship between effort and delivery time

There is a common myth that is still believed by many managers who are responsible for software development projects: "If we fall behind schedule, we can always add more programmers and catch up later in the project." Unfortunately, adding people late in a project often has a disruptive effect on the

project, causing schedules to slip even further. The people who are added must learn the system, and the people who teach them are the same people who were doing the work. While teaching, no work is done, and the project falls further behind. In addition to the time it takes to learn the system, more people increase the number of communication paths and the complexity of communication throughout a project. Although communication is absolutely essential to successful software development, every new communication path requires additional effort and therefore additional time. Over the years, empirical data and theoretical analysis have demonstrated that project schedules are elastic. That is, it is possible to compress a desired project completion date (by adding additional resources) to some extent. It is also possible to extend a completion date (by reducing the number of resources). The Putnam-Norden-Rayleigh (PNR) Curve5 provides an indication of the relationship between effort applied and delivery time for a software project. A version of the curve, representing project effort as a function of delivery time, is shown in Figure 27.1. The curve indicates a minimum value to that indicates the least cost for delivery (i.e., the delivery time that will result in the least effort expended). As we move left of to (i.e., as we try to accelerate delivery), the curve rises nonlinearly. As an example, we assume that a project team has estimated a level of effort Ed will be required to achieve a nominal delivery time td that is optimal in terms of 726 PART FOUR MANAGING SOFTWARE PROJECTS If you must add people to a late project, be sure that you've assigned them work that is highly compartmentalized.

schedule and available resources. Although it is possible to accelerate delivery, the curve rises very sharply to the left of td. In fact, the PNR curve indicates that the project delivery time cannot be compressed much beyond 0.75td. If we attempt further compression, the project moves into "the impossible region" and risk of failure becomes very high. The PNR curve also indicates that the lowest cost delivery option, to 2td. The implication here is that delaying project delivery can reduce costs significantly. Of course, this must be weighed against the business cost associated with the delay. The software equation [Put92] introduced in Chapter 26 is derived from the PNR curve and demonstrates the highly nonlinear relationship between chronological time to complete a project and human effort applied to the project. The number of delivered lines of code (source statements), L, is related to effort and development time by the equation: L P E1/3t 4/3 where E is development effort in person-months, P is a productivity parameter that reflects a variety of factors that lead to high-quality software engineering work (typical values for P range between 2000 and 12,000), and t is the project duration in calendar months. Rearranging this software equation, we can arrive at an expression for development effort E: E (27.1) where E is the effort expended (in person-years) over the entire life cycle for software development and maintenance and t is the development time in years. The equation for development effort can be related to development cost by the inclusion of a burdened labor rate factor ($/person-year). This leads to some interesting results. Consider a complex, real-time software project estimated at 33,000 LOC, 12 person-years of effort. If eight people are assigned to the project team, the project can be completed in approximately 1.3 years. If, however, we extend the end date to 1.75 years, the highly nonlinear nature of the model described in Equation (27.1) yields: E ~ 3.8 person-years This implies that, by extending the end date by six months, we can reduce the number of people from eight to four! The validity of such results is open to debate, but the implication is clear: benefit can be gained by using fewer people over a somewhat longer time span to accomplish the same objective. 27.2.3 Effort Distribution Each of the software project estimation techniques discussed in Chapter 26 leads to

estimates of work units (e.g., person-months) required to complete software development. A recommended distribution of effort across the software process is often referred to as the 40–20–40 rule. Forty percent of all effort is allocated to frontend analysis and design. A similar percentage is applied to back-end testing. You can correctly infer that coding (20 percent of effort) is deemphasized. This effort distribution should be used as a guideline only.6 The characteristics of each project dictate the distribution of effort. Work expended on project planning rarely accounts for more than 2 to 3 percent of effort, unless the plan commits an organization to large expenditures with high risk. Customer communication and requirements analysis may comprise 10 to 25 percent of project effort. Effort expended on analysis or prototyping should increase in direct proportion with project size and complexity. A range of 20 to 25 percent of effort is normally applied to software design. Time expended for design review and subsequent iteration must also be considered. Because of the effort applied to software design, code should follow with relatively little difficulty. A range of 15 to 20 percent of overall effort can be achieved. Testing and subsequent debugging can account for 30 to 40 percent of software development effort. The criticality of the software often dictates the amount of testing that is required. If software is human rated (i.e., software failure can result in loss of life), even higher percentages are typical.

QUALITY CONCEPTS

QUALITY

Quality . . . you know what it is, yet you don't know what it is. But that's self-contradictory. But some things are better than others; that is, they have more quality. But when you try to say what the quality is, apart from the things that have it, it all goes poof! There's nothing to talk about. But if you can't say what Quality is, how do you know what it is, or how do you know that it even exists? If no one knows what it is, then for all practical purposes it doesn't exist at all. But for all practical purposes it really does exist. What else are the grades based on? Why else would people pay fortunes for some things and throw others in the trash pile? Obviously some things are better than others . . . but what's the betterness? . . . So round and round you go, spinning mental wheels and nowhere finding anyplace to get traction. What the hell is Quality? What is it?

user satisfaction =compliant product+ good quality +delivery within budget and schedule

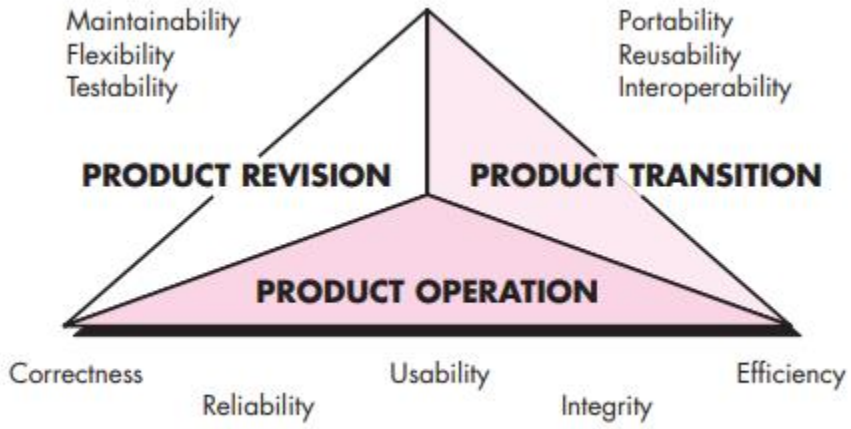
SOFTWARE QUALITY:

Even the most jaded software developers will agree that high-quality software is an important goal. But how do we define software quality? In the most general sense, software quality can be defined1 as: An effective software process applied in a manner that creates a useful product that provides measurable value for those who produce it and those who use it. There is little question that the preceding definition could be modified or extended and debated endlessly. For the purposes of this book, the definition serves to emphasize three important points: 1. An effective software process establishes the infrastructure that supports any effort at building a high-quality software product. The management aspects of process create the checks and balances that help avoid project chaos—a key contributor to poor quality. Software engineering practices allow the developer to analyze the problem and design a

solid solution—both critical to building high-quality software. Finally, umbrella activities such as change management and technical reviews have as much to do with quality as any other part of software engineering practice. 2. A useful product delivers the content, functions, and features that the end user desires, but as important, it delivers these assets in a reliable, error-free way. A useful product always satisfies those requirements that have been explicitly stated by stakeholders. In addition, it satisfies a set of implicit requirements (e.g., ease of use) that are expected of all high-quality software. 3. By adding value for both the producer and user of a software product, highquality software provides benefits for the software organization and the enduser community. The software organization gains added value because high-quality software requires less maintenance effort, fewer bug fixes, and reduced customer support. This enables software engineers to spend more time creating new applications and less on rework. The user community gains added value because the application provides a useful capability in a way that expedites some business process. The end result is (1) greater software product revenue, (2) better profitability when an application supports a business process, and/or (3) improved availability of information that is crucial for the business. 14.2.1 Garvin's Quality Dimensions David Garvin [Gar87] suggests that quality should be considered by taking a multidimensional viewpoint that begins with an assessment of conformance and terminates with a transcendental (aesthetic) view. Although Garvin's eight dimensions of quality were not developed specifically for software, they can be applied when software quality is considered: Performance quality. Does the software deliver all content, functions, and features that are specified as part of the requirements model in a way that provides value to the end user? Feature quality. Does the software provide features that surprise and delight first-time end users? Reliability. Does the software deliver all features and capability without failure? Is it available when it is needed? Does it deliver functionality that is error-free? Conformance. Does the software conform to local and external software standards that are relevant to the application? Does it conform to de facto design and coding conventions? For example, does the user interface conform to accepted design rules for menu selection or data input? Durability. Can the software be maintained (changed) or corrected (debugged) without the inadvertent generation of unintended side effects? Will changes cause the error rate or reliability to degrade with time? Serviceability. Can the software be maintained (changed) or corrected (debugged) in an acceptably short time period? Can support staff acquire all information they need to make changes or correct defects?

between something that can go wrong and something that can't possibly go wrong is that when something that can't possibly go wrong goes wrong it usually turns out to be impossible to get at or repair." Aesthetics. There's no question that each of us has a different and very subjective vision of what is aesthetic. And yet, most of us would agree that an aesthetic entity has a certain elegance, a unique flow, and an obvious "presence" that are hard to quantify but are evident nonetheless. Aesthetic software has these characteristics. Perception. In some situations, you have a set of prejudices that will influence your perception of quality. For example, if you are introduced to a software product that was built by a vendor who has produced poor quality in the past, your guard will be raised and your perception of the current software product quality might be influenced negatively. Similarly, if a vendor has an excellent reputation, you may perceive quality, even when it does not really exist. Garvin's quality dimensions provide you with a "soft" look at software quality. Many (but not all) of these

dimensions can only be considered subjectively. For this reason, you also need a set of “hard” quality factors that can be categorized in two broad groups: (1) factors that can be directly measured (e.g., defects uncovered during testing) and (2) factors that can be measured only indirectly (e.g., usability or maintainability). In each case measurement must occur. You should compare the software to some datum and arrive at an indication of quality. 14.2.2 McCall’s Quality Factors McCall, Richards, and Walters [McC77] propose a useful categorization of factors that affect software quality. These software quality factors, shown in Figure 14.1, focus on three important aspects of a software product: its operational characteristics, its ability to undergo change, and its adaptability to new environments. Referring to the factors noted in Figure 14.1, McCall and his colleagues provide the following descriptions: Correctness. The extent to which a program satisfies its specification and fulfills the customer’s mission objectives. Reliability. The extent to which a program can be expected to perform its intended function with required precision. [It should be noted that other, more complete definitions of reliability have been proposed (see Chapter 25).] Efficiency. The amount of computing resources and code required by a program to perform its function. Integrity. Extent to which access to software or data by unauthorized persons can be controlled. Usability. Effort required to learn, operate, prepare input for, and interpret output of a program.



Maintainability. Effort required to locate and fix an error in a program. [This is a very limited definition.] Flexibility. Effort required to modify an operational program. Testability. Effort required to test a program to ensure that it performs its intended function. Portability. Effort required to transfer the program from one hardware and/or software system environment to another. Reusability. Extent to which a program [or parts of a program] can be reused in other applications—related to the packaging and scope of the functions that the program performs. Interoperability. Effort required to couple one system to another. It is difficult, and in some cases impossible, to develop direct measures2 of these quality factors. In fact, many of the metrics defined by McCall et al. can be measured only indirectly. However, assessing the quality of an application using these factors will provide you with a solid indication of software quality. 14.2.3 ISO 9126 Quality Factors The ISO 9126 standard was developed in an attempt to identify the key quality attributes for computer software. The standard identifies six key quality attributes: Functionality. The degree to which the software satisfies stated needs as indicated by

the following subattributes: suitability, accuracy, interoperability, compliance, and security. Reliability. The amount of time that the software is available for use as indicated by the following subattributes: maturity, fault tolerance, recoverability.

Usability. The degree to which the software is easy to use as indicated by the following subattributes: understandability, learnability, operability. Efficiency. The degree to which the software makes optimal use of system resources as indicated by the following subattributes: time behavior, resource behavior. Maintainability. The ease with which repair may be made to the software as indicated by the following subattributes: analyzability, changeability, stability, testability. Portability. The ease with which the software can be transposed from one environment to another as indicated by the following subattributes: adaptability, installability, conformance, replaceability. Like other software quality factors discussed in the preceding subsections, the ISO 9126 factors do not necessarily lend themselves to direct measurement. However, they do provide a worthwhile basis for indirect measures and an excellent checklist for assessing the quality of a system. 14.2.4 Targeted Quality Factors The quality dimensions and factors presented in Sections 14.2.1 and 14.2.2 focus on the software as a whole and can be used as a generic indication of the quality of an application. A software team can develop a set of quality characteristics and associated questions that would probe3 the degree to which each factor has been satisfied. For example, McCall identifies usability as an important quality factor. If you were asked to review a user interface and assess its usability, how would you proceed? You might start with the subattributes suggested by McCall—understandability, learnability, and operability—but what do these mean in a pragmatic sense? To conduct your assessment, you'll need to address specific, measurable (or at least, recognizable) attributes of the interface. For example [Bro03]: Intuitiveness. The degree to which the interface follows expected usage patterns so that even a novice can use it without significant training. • Is the interface layout conducive to easy understanding? • Are interface operations easy to locate and initiate? • Does the interface use a recognizable metaphor? • Is input specified to economize key strokes or mouse clicks? • Does the interface follow the three golden rules? (Chapter 11) • Do aesthetics aid in understanding and usage?

Efficiency. The degree to which operations and information can be located or initiated. • Does the interface layout and style allow a user to locate operations and information efficiently? • Can a sequence of operations (or data input) be performed with an economy of motion? • Are output data or content presented so that it is understood immediately? • Have hierarchical operations been organized in a way that minimizes the depth to which a user must navigate to get something done? Robustness. The degree to which the software handles bad input data or inappropriate user interaction. • Will the software recognize the error if data at or just outside prescribed boundaries is input? More importantly, will the software continue to operate without failure or degradation? • Will the interface recognize common cognitive or manipulative mistakes and explicitly guide the user back on the right track? • Does the interface provide useful diagnosis and guidance when an error condition (associated with software functionality) is uncovered? Richness. The degree to which the interface provides a rich feature set. • Can the interface be customized to the specific needs of a user? • Does the interface provide a macro capability that enables a user to identify a sequence of common operations with a single action or command? As the interface design is developed, the software team would review the design prototype

and ask the questions noted. If the answer to most of these questions is “yes,” it is likely that the user interface exhibits high quality. A collection of questions similar to these would be developed for each quality factor to be assessed. 14.2.5 The Transition to a Quantitative View In the preceding subsections, I have presented a variety of qualitative factors for the “measurement” of software quality. The software engineering community strives to develop precise measures for software quality and is sometimes frustrated by the subjective nature of the activity. Cavano and McCall [Cav78] discuss this situation: The determination of quality is a key factor in every day events—wine tasting contests, sporting events [e.g., gymnastics], talent contests, etc. In these situations, quality is judged in the most fundamental and direct manner: side by side comparison of objects under identical conditions and with predetermined concepts. The wine may be judged according to clarity, color, bouquet, taste, etc. However, this type of judgment is very subjective; to have any value at all, it must be made by an expert.

Subjectivity and specialization also apply to determining software quality. To help solve this problem, a more precise definition of software quality is needed as well as a way to derive quantitative measurements of software quality for objective analysis. . . . Since there is no such thing as absolute knowledge, one should not expect to measure software quality exactly, for every measurement is partially imperfect. Jacob Bronkowski described this paradox of knowledge in this way: “Year by year we devise more precise instruments with which to observe nature with more fineness. And when we look at the observations we are discomfited to see that they are still fuzzy, and we feel that they are as uncertain as ever.” In Chapter 23, I’ll present a set of software metrics that can be applied to the quantitative assessment of software quality. In all cases, the metrics represent indirect measures; that is, we never really measure quality but rather some manifestation of quality. The complicating factor is the precise relationship between the variable that is measured and the quality of software.

Software quality doesn’t just appear. It is the result of good project management and solid software engineering practice. Management and practice are applied within the context of four broad activities that help a software team achieve high software quality: software engineering methods, project management techniques, quality control actions, and software quality assurance. 14.4.1 Software Engineering Methods If you expect to build high-quality software, you must understand the problem to be solved. You must also be capable of creating a design that conforms to the problem while at the same time exhibiting characteristics that lead to software that exhibits the quality dimensions and factors discussed in Section 14.2. In Part 2 of this book, I presented a wide array of concepts and methods that can lead to a reasonably complete understanding of the problem and a comprehensive design that establishes a solid foundation for the construction activity. If you apply those concepts and adopt appropriate analysis and design methods, the likelihood of creating high-quality software will increase substantially. 14.4.2 Project Management Techniques The impact of poor management decisions on software quality has been discussed in Section 14.3.6. The implications are clear: if (1) a project manager uses estimation to verify that delivery dates are achievable, (2) schedule dependencies are understood and the team resists the temptation to use short cuts, (3) risk planning is conducted so problems do not breed chaos, software quality will be affected in a positive way. In addition, the project plan should include explicit techniques for quality and change management. Techniques that lead to good project management practices are discussed in Part 4 of this book. 14.4.3 Quality Control Quality

control encompasses a set of software engineering actions that help to ensure that each work product meets its quality goals. Models are reviewed to ensure that they are complete and consistent. Code may be inspected in order to uncover and correct errors before testing commences. A series of testing steps is applied to uncover errors in processing logic, data manipulation, and interface communication. A combination of measurement and feedback allows a software team to tune the process when any of these work products fail to meet quality goals. Quality control activities are discussed in detail throughout the remainder of Part 3 of this book.

Quality Assurance Quality assurance establishes the infrastructure that supports solid software engineering methods, rational project management, and quality control actions—all pivotal if you intend to build high-quality software. In addition, quality assurance consists of a set of auditing and reporting functions that assess the effectiveness and completeness of quality control actions. The goal of quality assurance is to provide management and technical staff with the data necessary to be informed about product quality, thereby gaining insight and confidence that actions to achieve product quality are working. Of course, if the data provided through quality assurance identifies problems, it is management's responsibility to address the problems and apply the necessary resources to resolve quality issues. Software quality assurance is discussed in detail in Chapter 16.

A FORMAL TECHNICAL REVIEWS

A formal technical review (FTR) is a software quality control activity performed by software engineers (and others). The objectives of an FTR are: (1) to uncover errors in function, logic, or implementation for any representation of the software; (2) to verify that the software under review meets its requirements; (3) to ensure that the software has been represented according to predefined standards; (4) to achieve software that is developed in a uniform manner; and (5) to make projects more manageable. In addition, the FTR serves as a training ground, enabling junior engineers to observe different approaches to software analysis, design, and implementation. The FTR also serves to promote backup and continuity because a number of people become familiar with parts of the software that they may not have otherwise seen. The FTR is actually a class of reviews that includes walkthroughs and inspections. Each FTR is conducted as a meeting and will be successful only if it is properly planned, controlled, and attended. In the sections that follow, guidelines similar to those for a walkthrough are presented as a representative formal technical review. If you have interest in software inspections, as well as additional information on walkthroughs, see [Rad02], [Wie02], or [Fre90]. 15.6.1 The Review Meeting Regardless of the FTR format that is chosen, every review meeting should abide by the following constraints: • Between three and five people (typically) should be involved in the review. • Advance preparation should occur but should require no more than two hours of work for each person. • The duration of the review meeting should be less than two hours. Given these constraints, it should be obvious that an FTR focuses on a specific (and small) part of the overall software. For example, rather than attempting to review an entire design, walkthroughs are conducted for each component or small group of components. By narrowing the focus, the FTR has a higher likelihood of uncovering errors. The focus of the FTR is on a work product (e.g., a portion of a requirements model, a detailed component design, source code for a component). The individual who has developed the work product—the producer—informs the project leader that the work product is complete and that a review is required. The project

leader contacts a review leader, who evaluates the product for readiness, generates copies of product materials, and distributes them to two or three reviewers for advance preparation. Each reviewer is expected to spend between one and two hours reviewing the product, making notes, and otherwise becoming familiar with the work. Concurrently, the review leader also reviews the product and establishes an agenda for the review meeting, which is typically scheduled for the next day.

The review meeting is attended by the review leader, all reviewers, and the producer. One of the reviewers takes on the role of a recorder, that is, the individual who records (in writing) all important issues raised during the review. The FTR begins with an introduction of the agenda and a brief introduction by the producer. The producer then proceeds to "walk through" the work product, explaining the material, while reviewers raise issues based on their advance preparation. When valid problems or errors are discovered, the recorder notes each. At the end of the review, all attendees of the FTR must decide whether to: (1) accept the product without further modification, (2) reject the product due to severe errors (once corrected, another review must be performed), or (3) accept the product provisionally (minor errors have been encountered and must be corrected, but no additional review will be required). After the decision is made, all FTR attendees complete a sign-off, indicating their participation in the review and their concurrence with the review team's findings. 15.6.2 Review Reporting and Record Keeping During the FTR, a reviewer (the recorder) actively records all issues that have been raised. These are summarized at the end of the review meeting, and a review issues list is produced. In addition, a formal technical review summary report is completed. A review summary report answers three questions: 1. What was reviewed? 2. Who reviewed it? 3. What were the findings and conclusions? The review summary report is a single page form (with possible attachments). It becomes part of the project historical record and may be distributed to the project leader and other interested parties. The review issues list serves two purposes: (1) to identify problem areas within the product and (2) to serve as an action item checklist that guides the producer as corrections are made. An issues list is normally attached to the summary report. You should establish a follow-up procedure to ensure that items on the issues list have been properly corrected. Unless this is done, it is possible that issues raised can "fall between the cracks." One approach is to assign the responsibility for follow-up to the review leader. 15.6.3 Review Guidelines Guidelines for conducting formal technical reviews must be established in advance, distributed to all reviewers, agreed upon, and then followed. A review that is uncontrolled can often be worse than no review at all. The following represents a minimum set of guidelines for formal technical reviews: 1. Review the product, not the producer. An FTR involves people and egos. Conducted properly, the FTR should leave all participants with a warm feeling of accomplishment. Conducted improperly, the FTR can take on the aura of an inquisition. Errors should be pointed out gently; the tone of the meeting should be loose and constructive; the intent should not be to embarrass or belittle. The review leader should conduct the review meeting to ensure that the proper tone and attitude are maintained and should immediately halt a review that has gotten out of control. 2. Set an agenda and maintain it. One of the key maladies of meetings of all types is drift. An FTR must be kept on track and on schedule. The review leader is chartered with the responsibility for maintaining the meeting schedule and should not be afraid to nudge people when drift sets in. 3. Limit debate and rebuttal. When an issue is raised by a reviewer, there may not be universal agreement on its impact. Rather than spending time debating the question, the issue should be recorded for further discussion

off-line. 4. Enunciate problem areas, but don't attempt to solve every problem noted. A review is not a problem-solving session. The solution of a problem can often be accomplished by the producer alone or with the help of only one other individual. Problem solving should be postponed until after the review meeting. 5. Take written notes. It is sometimes a good idea for the recorder to make notes on a wall board, so that wording and priorities can be assessed by other reviewers as information is recorded. Alternatively, notes may be entered directly into a notebook computer. 6. Limit the number of participants and insist upon advance preparation. Two heads are better than one, but 14 are not necessarily better than 4. Keep the number of people involved to the necessary minimum. However, all review team members must prepare in advance. Written comments should be solicited by the review leader (providing an indication that the reviewer has reviewed the material). 7. Develop a checklist for each product that is likely to be reviewed. A checklist helps the review leader to structure the FTR meeting and helps each reviewer to focus on important issues. Checklists should be developed for analysis, design, code, and even testing work products. 8. Allocate resources and schedule time for FTRs. For reviews to be effective, they should be scheduled as tasks during the software process. In addition, time should be scheduled for the inevitable modifications that will occur as the result of an FTR. 9. Conduct meaningful training for all reviewers. To be effective all review participants should receive some formal training. The training should stress both process-related issues and the human psychological side of reviews. Freedman and Weinberg [Fre90] estimate a one-month learning curve for every 20 people who are to participate effectively in reviews. 428 PART THREE QUALITY MANAGEMENT uote: "A meeting is too often an event in which minutes are taken and hours are wasted." Author unknown uote: "It is one of the most beautiful compensations of life, that no man can sincerely try to help another without helping himself." Ralph Waldo Emerson 10. Review your early reviews. Debriefing can be beneficial in uncovering problems with the review process itself. The very first product to be reviewed should be the review guidelines themselves. Because many variables (e.g., number of participants, type of work products, timing and length, specific review approach) have an impact on a successful review, a software organization should experiment to determine what approach works best in a local context. 15.6.4 Sample-Driven Reviews In an ideal setting, every software engineering work product would undergo a formal technical review. In the real word of software projects, resources are limited and time is short. As a consequence, reviews are often skipped, even though their value as a quality control mechanism is recognized. Thelin and his colleagues [The01] suggest a sample-driven review process in which samples of all software engineering work products are inspected to determine which work products are most error prone. Full FTR resources are then focused only on those work products that are likely (based on data collected during sampling) to be error prone. To be effective, the sample-driven review process must attempt to quantify those work products that are primary targets for full FTRs. To accomplish this, the following steps are suggested [The01]: 1. Inspect a fraction ai of each software work product i. Record the number of faults fi found within ai . 2. Develop a gross estimate of the number of faults within work product i by multiplying fi by 1/ai . 3. Sort the work products in descending order according to the gross estimate of the number of faults in each. 4. Focus available review resources on those work products that have the highest estimated number of faults. The fraction of the work product that is sampled must be representative of the work product as a whole and large enough to be meaningful to the reviewers who do the sampling. As ai increases, the likelihood that the sample is a valid representation of the work product also increases. However, the resources

required to do sampling also increase. A software engineering team must establish the best value for ai for the types of work products produced.