CHAPTER

3 PROJECT MANAGEMENT CONCEPTS

KEY CONCEPTS

critical practices . . 74
common process framework 70
coordination 65
problem decomposition . . . 67
process decomposition . . . 70
scope 67
software team . . 60
team leader 59
team structure . . . 60
team toxicity 63
W^{5}HH principle . . 73

In the preface to his book on software project management, Meiler Page-Jones [PAG85] makes a statement that can be echoed by many software engineering consultants:

> I've visited dozens of commercial shops, both good and bad, and I've observed scores of data processing managers, again, both good and bad. Too often, I've watched in horror as these managers futilely struggled through nightmarish projects, squirmed under impossible deadlines, or delivered systems that outraged their users and went on to devour huge chunks of maintenance time.

What Page-Jones describes are symptoms that result from an array of management and technical problems. However, if a post mortem were to be conducted for every project, it is very likely that a consistent theme would be encountered: project management was weak.

In this chapter and the six that follow, we consider the key concepts that lead to effective software project management. This chapter considers basic software project management concepts and principles. Chapter 4 presents process and project metrics, the basis for effective management decision making. The techniques that are used to estimate cost and resource requirements and establish an effective project plan are discussed in Chapter 5. The man-

QUICK LOOK

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 an operational implementation.

Who does it? Everyone "manages" to some extent, but the scope of management activities varies with the person doing it. 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 the 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 must occur so that product scope and requirements are understood. A process must be selected that is appropriate for the people and the product. The project must be planned by estimating effort and calendar time to accomplish work tasks: defining work products, establishing quality checkpoints, and

QUICK LOOK

establishing 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.

agement activities that lead to effective risk monitoring, mitigation, and management are presented in Chapter 6. Chapter 7 discusses the activities that are required to define project tasks and establish a workable project schedule. Finally, Chapters 8 and 9 consider techniques for ensuring quality as a project is conducted and controlling changes throughout the life of an application.

3.1 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 customer communication early in the evolution of a project 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 product.

3.1.1 The People

Quote:

"There exists enormous variability in the ability of different people to perform programming tasks."

Bill Curtis

The cultivation of motivated, highly skilled software people has been discussed since the 1960s (e.g., [COU80], [WIT94], [DEM98]). In fact, the "people factor" is so important that the Software Engineering Institute has developed a *people management capability maturity model* (PM-CMM), "to enhance the readiness of software organizations to undertake increasingly complex applications by helping to attract, grow, motivate, deploy, and retain the talent needed to improve their software development capability" [CUR94].

The people management maturity model defines the following key practice areas for software people: recruiting, selection, performance management, training, compensation, career development, organization and work design, and team/culture development. Organizations that achieve high levels of maturity in the people management area have a higher likelihood of implementing effective software engineering practices.

The PM-CMM is a companion to the software capability maturity model (Chapter 2) that guides organizations in the creation of a mature software process. Issues

associated with people management and structure for software projects are considered later in this chapter.

3.1.2 The Product

XRef

A taxonomy of application areas that spawn software "products" is discussed in Chapter 1.

Before a project can be planned, product[1] 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.

The software developer and customer must meet to define product objectives and scope. In many cases, this activity begins as part of the system engineering or business process engineering (Chapter 10) and continues as the first step in software requirements analysis (Chapter 11). Objectives identify the overall goals for the product (from the customer's point 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.

3.1.3 The Process

Framework activities are populated with tasks, milestones, work products, and quality assurance points.

A software process (Chapter 2) 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.

3.1.4 The Project

We conduct planned and controlled software projects for one primary reason—it is the only known way to manage complexity. And yet, we still struggle. In 1998, industry data indicated that 26 percent of software projects failed outright and 46 percent experienced cost and schedule overruns [REE99]. Although the success rate for

---

1 In this context, the term *product* is used to encompass any software that is to be built at the request of others. It includes not only software products but also computer-based systems, embedded software, and problem-solving software (e.g., programs for engineering/scientific problem solving).

software projects has improved somewhat, our project failure rate remains higher than it should be.[2]

In order 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 3.5 and in the chapters that follow.

3.2 PEOPLE

In a study published by the IEEE [CUR88], the engineering vice presidents of three major technology companies were asked the most important contributor to a successful software project. They answered in the following way:

Quote:

"Companies that sensibly manage their investment in people will prosper in the long run."

Tom DeMarco & Tim Lister

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 we examine the players who participate in the software process and the manner in which they are organized to perform effective software engineering.

3.2.1 The Players

The software process (and every software project) is populated by players who can be categorized into one of five constituencies:

1. **Senior managers** who define the business issues that often have significant influence on the project.

2 Given these statistics, it's reasonable to ask how the impact of computers continues to grow exponentially and the software industry continues to post double digit sales growth. Part of the answer, I think, is that a substantial number of these "failed" projects are ill-conceived in the first place. Customers lose interest quickly (because what they requested wasn't really as important as they first thought), and the projects are cancelled.

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. 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.

3.2.2 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]

? What do we look for when we select someone to lead a software project?

In an excellent book of technical leadership, Jerry Weinberg [WEI86] suggests a 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.

Quote:

"In simplest terms, a leader is one who knows where he wants to go, and gets up, and goes."
John Erskine

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. To optimize the productivity of a project team, a manager must reward initiative and accomplishment and demonstrate through his 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.

A software wizard may not have the temperament or desire to be a team leader. Don't force the wizard to become one.

3.2.3 The Software Team

"Not every group is a team, and not every team is effective."
Glenn Parker

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 following options are available for applying human resources to a project that will require n people working for k years:

1. n individuals are assigned to m different functional tasks, relatively little combined work occurs; coordination is the responsibility of a software manager who may have six other projects to be concerned with.
2. n individuals are assigned to m different functional tasks ($m < n$) so that informal "teams" are established; an ad hoc team leader may be appointed; coordination among teams is the responsibility of a software manager.
3. n individuals are organized into t teams; each team is assigned one or more functional tasks; each team has a specific structure that is defined for all teams working on a project; coordination is controlled by both the team and a software project manager.

How should a software team be organized?

Although it is possible to voice arguments for and against each of these approaches, a growing body of evidence indicates that a formal team organization (option 3) is most productive.

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] suggests three generic team organizations:

Democratic decentralized (DD). This software engineering team has no permanent leader. Rather, "task coordinators are appointed for short durations and then replaced by others who may coordinate different tasks." Decisions on problems and approach are made by group consensus. Communication among team members is horizontal.

Controlled decentralized (CD). This software engineering team has a defined leader who coordinates specific tasks and secondary leaders that have responsibility for subtasks. Problem solving remains a group activity, but implementation of solutions is partitioned among subgroups by the team leader. Communication among subgroups and individuals is horizontal. Vertical communication along the control hierarchy also occurs.

Controlled Centralized (CC). Top-level problem solving and internal team coordination are managed by a team leader. Communication between the leader and team members is vertical.

Mantei [MAN81] describes seven project factors that should be considered when planning the structure of software engineering teams:

? What factors should we consider when structuring a software team?

- The difficulty of the problem to be solved.
- The size of the resultant program(s) in lines of code or function points (Chapter 4).
- The time that the team will stay together (team lifetime).
- The degree to which the problem can be modularized.
- The required quality and reliability of the system to be built.
- The rigidity of the delivery date.
- The degree of sociability (communication) required for the project.

Because a centralized structure completes tasks faster, it is the most adept at handling simple problems. Decentralized teams generate more and better solutions than individuals. Therefore such teams have a greater probability of success when working on difficult problems. Since the CD team is centralized for problem solving, either a CD or CC team structure can be successfully applied to simple problems. A DD structure is best for difficult problems.

It's often better to have a few small, well-focused teams than a single large team.

Because the performance of a team is inversely proportional to the amount of communication that must be conducted, very large projects are best addressed by teams with a CC or CD structures when subgrouping can be easily accommodated.

The length of time that the team will "live together" affects team morale. It has been found that DD team structures result in high morale and job satisfaction and are therefore good for teams that will be together for a long time.

The DD team structure is best applied to problems with relatively low modularity, because of the higher volume of communication needed. When high modularity is possible (and people can do their own thing), the CC or CD structure will work well.

CC and CD teams have been found to produce fewer defects than DD teams, but these data have much to do with the specific quality assurance activities that are applied by the team. Decentralized teams generally require more time to complete a project than a centralized structure and at the same time are best when high sociability is required.

Constantine [CON93] suggests four "organizational paradigms" for software engineering teams:

Quote:

"Working with people is difficult, but not impossible."

Peter Drucker

1. A *closed paradigm* structures a team along a traditional hierarchy of authority (similar to a CC team). 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. The *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. The *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. The *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, the earliest software team organization was a controlled centralized (CD) 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.

XRef

The role of the librarian exists regardless of team structure. See Chapter 9 for details.

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*. The librarian serves many teams and performs the following functions: maintains and controls all elements of the software configuration (i.e., documentation, source listings, data, storage media); helps collect and format software productivity data; catalogs and indexes reusable software compo-

nents; and assists the teams in research, evaluation, and document preparation. The importance of a librarian cannot be overemphasized. The librarian acts as a controller, coordinator, and potentially, an evaluator of the software configuration.

A variation on the democratic decentralized team has been proposed by Constantine [CON93], who advocates teams 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 high-performance team must be a central goal of a software engineering organization. To achieve a high-performance team:

Quote:

"No matter what the problem is, it's always a people problem."

Jerry Weinberg

- 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:

> We tend to use the word *team* fairly loosely in the business world, calling any group of people assigned to work together a "team." But many of these groups just don't seem like teams. They don't have a common definition of success or any identifiable team spirit. What is missing is a phenomenon that we call *jell.*
>
> A jelled team is a group of people so strongly knit that the whole is greater than the sum of the parts . . .
>
> Once a team begins to jell, the probability of success goes way up. The team can become unstoppable, a juggernaut for success . . . They don't need to be managed in the traditional way, and they certainly don't need to be motivated. They've got momentum.

ADVICE

Jelled teams are the ideal, but they're not easy to achieve. At a minimum, be certain to avoid a "toxic environment."

DeMarco and Lister contend that members of jelled teams are significantly more productive and more motivated than average. They share a common goal, a common culture, and in many cases, a "sense of eliteness" that makes them unique.

But not all teams jell. In fact, many teams suffer from what Jackman calls "team toxicity" [JAC98]. She defines five factors that "foster a potentially toxic team environment":

1. A frenzied work atmosphere in which team members waste energy and lose focus on the objectives of the work to be performed.
2. High frustration caused by personal, business, or technological factors that causes friction among team members.
3. "Fragmented or poorly coordinated procedures" or a poorly defined or improperly chosen process model that becomes a roadblock to accomplishment.

4. Unclear definition of roles resulting in a lack of accountability and resultant finger-pointing.
5. "Continuous and repeated exposure to failure" that leads to a loss of confidence and a lowering of morale.

Jackman suggests a number of antitoxins that address these all-too-common problems.

? How do we avoid "toxins" that often infect a software team?

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. In addition, bad news should not be kept secret but rather, delivered to the team as early as possible (while there is still time to react in a rational and controlled manner).

Although frustration has many causes, software people often feel it when they lack the authority to control their situation. A software team can avoid frustration if it is given as much responsibility for decision making as possible. The more control over process and technical decisions given to the team, the less frustration the team members will feel.

An inappropriately chosen software process (e.g., unnecessary or burdensome work tasks or poorly chosen work products) can be avoided in two ways: (1) being certain that the characteristics of the software to be built conform to the rigor of the process that is chosen and (2) allowing the team to select the process (with full recognition that, once chosen, the team has the responsibility to deliver a high-quality product).

Quote:

"Do or do not; there is no try."

Yoda
(Star Wars)

The software project manager, working together with the team, should clearly refine roles and responsibilities before the project begins. The team itself should establish its own mechanisms for accountability (formal technical reviews[3] are an excellent way to accomplish this) and define a series of corrective approaches when a member of the team fails to perform.

Every software team experiences small failures. The key to avoiding an atmosphere of failure is to establish team-based techniques for feedback and problem solving. In addition, failure by any member of the team must be viewed as a failure by the team itself. This leads to a team-oriented approach to corrective action, rather than the finger-pointing and mistrust that grows rapidly on toxic teams.

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 introverted. 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

3 Formal technical reviews are discussed in detail in Chapter 8.

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.

3.2.4 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, a software engineering team 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 noninteractive 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.

Kraul and Streeter [KRA95] examine a collection of project coordination techniques that are categorized in the following manner:

Formal, impersonal approaches include software engineering documents and deliverables (including source code), technical memos, project milestones, schedules, and project control tools (Chapter 7), change requests and related documentation (Chapter 9), error tracking reports, and repository data (see Chapter 31).

Formal, interpersonal procedures focus on quality assurance activities (Chapter 8) applied to software engineering work products. These include status review meetings and design and code inspections.

Informal, interpersonal procedures include group meetings for information dissemination and problem solving and "collocation of requirements and development staff."

? How do we coordinate the actions of team members?

4 An excellent introduction to these issues as they relate to software project teams can be found in [FER98].

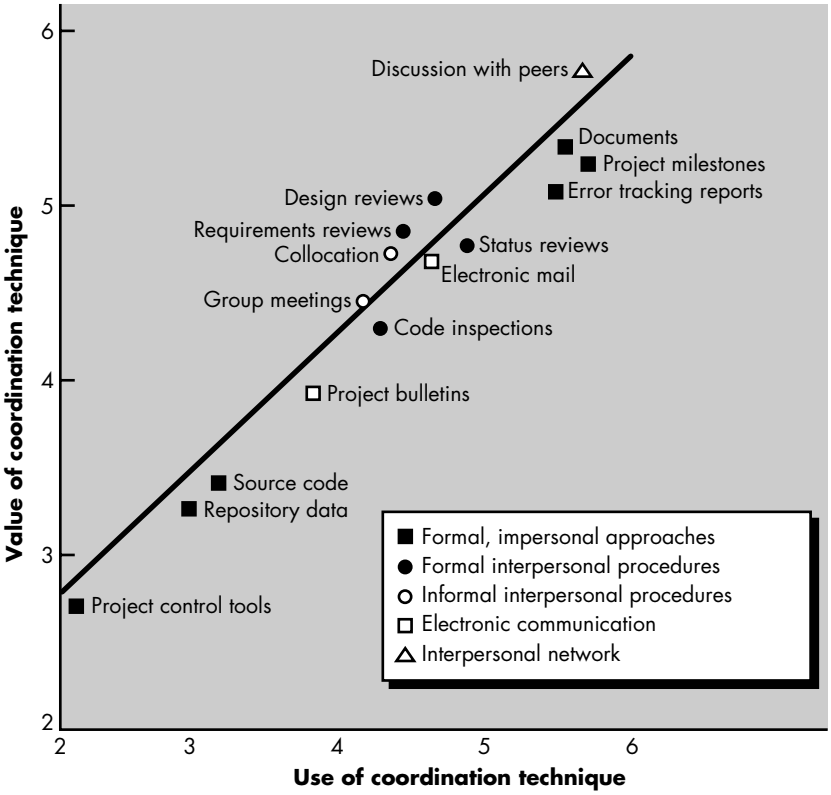


FIGURE 3.1 Value and Use of Coordination and Communication Techniques

Electronic communication encompasses electronic mail, electronic bulletin boards, and by extension, video-based conferencing systems.

Interpersonal networking includes informal discussions with team members and those outside the project who may have experience or insight that can assist team members.

To assess the efficacy of these techniques for project coordination, Kraul and Streeter studied 65 software projects involving hundreds of technical staff. Figure 3.1 (adapted from [KRA95]) expresses the value and use of the coordination techniques just noted. Referring to figure, the perceived value (rated on a seven point scale) of various coordination and communication techniques is plotted against their frequency of use on a project. Techniques that fall above the regression line were "judged to be relatively valuable, given the amount that they were used" [KRA95]. Techniques that fell below the line were perceived to have less value. It is interesting to note that interpersonal networking was rated the technique with highest coordination and communication value. It is also important to note that early software quality assurance mechanisms (requirements and design reviews) were perceived to have more value than later evaluations of source code (code inspections).

3.3 THE PRODUCT

A software project manager is confronted with a dilemma at the very beginning of a software engineering 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 months to complete. Worse, requirements may be fluid, changing regularly as the project proceeds. Yet, a plan is needed "now!"

Therefore, we 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.

3.3.1 Software Scope

The first software project management activity is the determination of *software scope.* Scope is defined by answering the following questions:

If you can't bound a characteristic of the software you intend to build, list the characteristic as a major project risk.

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 (Chapter 11) 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, size of mailing list, 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 C++) are described.

3.3.2 Problem Decomposition

In order to develop a reasonable project plan, you have to functionally decompose the problem to be solved.

Problem decomposition, sometimes called *partitioning* or *problem elaboration,* is an activity that sits at the core of software requirements analysis (Chapter 11). During the scoping activity no attempt is made to fully decompose the problem. Rather, decomposition is applied in two major areas: (1) the functionality 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 problems. 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 5). Because both cost and schedule estimates are functionally oriented, some degree of decomposition is often useful.

XRef

A useful technique for problem decomposition, called a *grammatical parse,* is presented in Chapter 12.

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 keyboard input, 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, document production, and the like). 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?

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?), and (4) 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.

3.4 THE PROCESS

The generic phases that characterize the software process—definition, development, and support—are applicable to all software. The problem is to select the process model that is appropriate for the software to be engineered by a project team. In Chapter 2, a wide array of software engineering paradigms were discussed:

- the linear sequential model
- the prototyping model
- the RAD model
- the incremental model
- the spiral model
- the WINWIN spiral model
- the component-based development model
- the concurrent development model
- the formal methods model
- the fourth generation techniques model

Once the process model is chosen, populate it with the minimum set of work tasks and work products that will result in a high-quality product—avoid process overkill!

The project manager 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 common 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 7.

3.4.1 Melding the Product and the Process

Project planning begins with the melding of the product and the process. Each function to be engineered by the software team must pass through the set of framework activities that have been defined for a software organization. Assume that the organization has adopted the following set of framework activities (Chapter 2):

Remember : framework activities are applied on every project—no exceptions.

- *Customer communication*—tasks required to establish effective requirements elicitation between developer and customer.
- *Planning*—tasks required to define resources, timelines, and other project-related information.
- *Risk analysis*—tasks required to assess both technical and management risks.
- *Engineering*—tasks required to build one or more representations of the application.
- *Construction and release*—tasks required to construct, test, install, and provide user support (e.g., documentation and training).
- *Customer evaluation*—tasks required to obtain customer feedback based on evaluation of the software representations created during the engineering activity and implemented during the construction activity.

Product and process decomposition occur simultaneously as the project plan evolves.

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 3.2 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 Chapters 5 and 7.

---

5 It should be noted that work tasks must be adapted to the specific needs of a project. Framework activities always remain the same, but work tasks will be selected based on a number of adaptation criteria. This topic is discussed further in Chapter 7 and at the SEPA Web site.

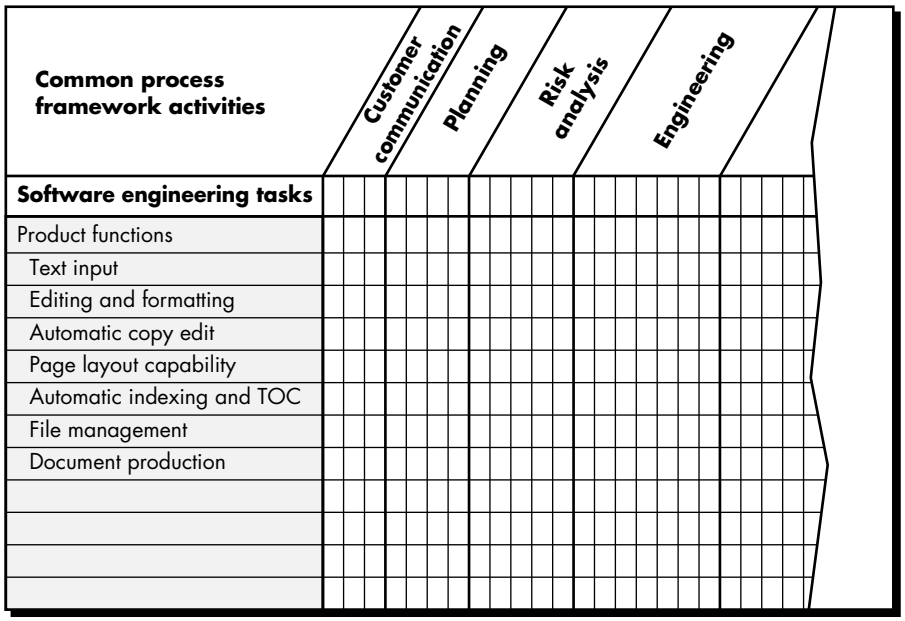


FIGURE 3.2 Melding the Problem and the Process

3.4.2 Process Decomposition

A software team should have a significant degree of flexibility in choosing the software engineering paradigm 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 very tight time constraints are imposed and the problem can be heavily compartmentalized, the RAD model is probably the right option. 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.[6]

Always apply the CPF, regardless of project size, criticality, or type. Work tasks may vary, but the CPF does not.

Once the process model has been chosen, the common process framework (CPF) is adapted to it. In every case, the CPF discussed earlier in this chapter—customer communication, planning, risk analysis, engineering, construction and release, customer evaluation—can be fitted to the paradigm. It will work for linear models, for iterative and incremental models, for evolutionary models, and even for concurrent or component assembly models. The CPF is invariant and serves as the basis for all software 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 CPF activity?" For example, a small,

6 Recall that project characteristics also have a strong bearing on the structure of the team that is to do the work. See Section 3.2.3.

relatively simple project might require the following work tasks for the *customer communication* activity:

1. Develop list of clarification issues.
2. Meet with customer 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.

Adaptable process model

Now, we 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 customer communication activity:

1. Review the customer request.
2. Plan and schedule a formal, facilitated meeting with the customer.
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, function, and behavioral features of the software.
7. Review each mini-spec for correctness, consistency, and lack of ambiguity.
8. Assemble the mini-specs into a scoping document.
9. Review the scoping document with all concerned.
10. Modify the scoping document as required.

Both projects perform the framework activity that we call "customer communication," but the first project team performed half as many software engineering work tasks as the second.

3.5 THE PROJECT

Quote:

"At least 7 of 10 signs of IS project failures are determined before a design is developed or a line of code is written . . ."

John Reel

In order to manage a successful software project, we must understand what can go wrong (so that problems can be avoided) and how to do it right. In an excellent paper on software projects, John Reel [REE99] defines ten 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 the other 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 preceeding 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 objects and expectations for everyone who will be involved in the project. It is reinforced by building the right team (Section 3.2.3) 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]
3. **Track progress.** For a software project, progress is tracked as work products (e.g., specifications, source code, sets of test cases) are produced and approved (using formal technical reviews) as part of a quality assurance activity. In addition, software process and project measures (Chapter 4) 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, decide to avoid custom interfaces when standard approaches are

A broad array of resources that can help both neophyte and experienced project managers can be found at **www.pmi.org, www.4pm.com,** and **www.projectmanagement.com**

7 The implication of this statement is that bureacracy is reduced to a minimum, extraneous meetings are eliminated, and dogmatic adherence to process and project rules is eliminated. The team should be allowed to do its thing.

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.

3.6 THE W^{5}HH PRINCIPLE

In an excellent paper on software process and projects, Barry Boehm [BOE96] states: "you need an organizing principle that scales down to provide simple [project] plans for simple projects." Boehm suggests an approach that addresses project objectives, milestones and schedules, responsibilities, management and technical approaches, and required resources. He calls it the WWWWWHH principle, after a series of questions that lead to a definition of key project characteristics and the resultant project plan:

What questions need to be answered in order to develop a project plan?

Why is the system being developed? The answer to this question enables all parties to assess the validity of business reasons for the software work. Stated in another way, does the business purpose justify the expenditure of people, time, and money?

What will be done, by when? The answers to these questions help the team to establish a project schedule by identifying key project tasks and the milestones that are required by the customer.

Who is responsible for a function? Earlier in this chapter, we noted that the role and responsibility of each member of the software team must be defined. The answer to this question helps accomplish this.

Where are they organizationally located? Not all roles and responsibilities reside within the software team itself. The customer, users, and other stakeholders also have responsibilities.

Software Project Plan

How will the job be done technically and managerially? Once product scope is established, a management and technical strategy for the project must be defined.

How much of each resource is needed? The answer to this question is derived by developing estimates (Chapter 5) based on answers to earlier questions.

Boehm's W^{5}HH principle is applicable regardless of the size or complexity of a software project. The questions noted provide an excellent planning outline for the project manager and the software team.

3.7 CRITICAL PRACTICES

The Airlie Council[8] 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]. In an effort to enable a software organization to determine whether a specific project has implemented critical practices, the Airlie Council has developed a set of "QuickLook" questions [AIR99] for a project:[9]

Airlie Project Quicklook

Formal risk management. What are the top ten risks for this project? For each of the risks, what is the chance that the risk will become a problem and what is the impact if it does?

Empirical cost and schedule estimation. What is the current estimated size of the application software (excluding system software) that will be delivered into operation? How was it derived?

Metric-based project management. Do you have in place a metrics program to give an early indication of evolving problems? If so, what is the current requirements volatility?

Earned value tracking. Do you report monthly earned value metrics? If so, are these metrics computed from an activity network of tasks for the entire effort to the next delivery?

Defect tracking against quality targets. Do you track and periodically report the number of defects found by each inspection (formal technical review) and execution test from program inception and the number of defects currently closed and open?

People-aware program management. What is the average staff turnover for the past three months for each of the suppliers/developers involved in the development of software for this system?

If a software project team cannot answer these questions or answers them inadequately, a thorough review of project practices is indicated. Each of the critical practices just noted is addressed in detail throughout Part Two of this book.

3.8 SUMMARY

Software project management is an umbrella activity within software engineering. It begins before any technical activity is initiated and continues throughout the definition, development, and support of computer software.

---

8 The Airlie Council is a team of software engineering experts chartered by the U.S. Department of Defense to help develop guidelines for best practices in software project management and software engineering.

9 Only those critical practices associated with "project integrity" are noted here. Other best practices will be discussed in later chapters.

Four P's have a substantial influence on software project management—people, product, process, and project. People must be organized into effective teams, motivated to do high-quality software work, and coordinated to achieve effective communication. The product requirements must be communicated from customer to developer, partitioned (decomposed) into their constituent parts, and positioned for work by the software team. The process must be adapted to the people and the problem. A common process framework is selected, an appropriate software engineering paradigm is applied, and a set of work tasks is chosen to get the job done. Finally, the project must be organized in a manner that enables the software team to succeed.

The pivotal element in all software projects is people. Software engineers can be organized in a number of different team structures that range from traditional control hierarchies to "open paradigm" teams. A variety of coordination and communication techniques can be applied to support the work of the team. In general, formal reviews and informal person-to-person communication have the most value for practitioners.

The project management activity encompasses measurement and metrics, estimation, risk analysis, schedules, tracking, and control. Each of these topics is considered in the chapters that follow.

REFERENCES

[AIR99] Airlie Council, "Performance Based Management: The Program Manager's Guide Based on the 16-Point Plan and Related Metrics," Draft Report, March 8, 1999.

[BAK72] Baker, F.T., "Chief Programmer Team Management of Production Programming," *IBM Systems Journal,* vol. 11, no. 1, 1972, pp. 56–73.

[BOE96] Boehm, B., "Anchoring the Software Process," *IEEE Software,* vol. 13, no. 4, July 1996, pp. 73–82.

[CON93] Constantine, L., "Work Organization: Paradigms for Project Management and Organization, *CACM,* vol. 36, no. 10, October 1993, pp. 34–43.

[COU80] Cougar, J. and R. Zawacki, *Managing and Motivating Computer Personnel,* Wiley, 1980.

[CUR88] Curtis, B. et al., "A Field Study of the Software Design Process for Large Systems," *IEEE Trans. Software Engineering,* vol. SE-31, no. 11, November 1988, pp. 1268–1287.

[CUR94] Curtis, B., et al., *People Management Capability Maturity Model,* Software Engineering Institute, 1994.

[DEM98] DeMarco, T. and T. Lister, *Peopleware,* 2nd ed., Dorset House, 1998.

[EDG95] Edgemon, J., "Right Stuff: How to Recognize It When Selecting a Project Manager," *Application Development Trends,* vol. 2, no. 5, May 1995, pp. 37–42.

[FER98] Ferdinandi, P.L., "Facilitating Communication," *IEEE Software,* September 1998, pp. 92–96.

[JAC98] Jackman, M., "Homeopathic Remedies for Team Toxicity," *IEEE Software,* July 1998, pp. 43–45.
[KRA95] Kraul, R. and L. Streeter, "Coordination in Software Development," *CACM,* vol. 38, no. 3, March 1995, pp. 69–81.
[MAN81] Mantei, M., "The Effect of Programming Team Structures on Programming Tasks," *CACM,* vol. 24, no. 3, March 1981, pp. 106–113.
[PAG85] Page-Jones, M., *Practical Project Management,* Dorset House, 1985, p. vii.
[REE99] Reel, J.S., "Critical Success Factors in Software Projects, *IEEE Software,* May, 1999, pp. 18–23.
[WEI86] Weinberg, G., *On Becoming a Technical Leader,* Dorset House, 1986.
[WIT94] Whitaker, K., *Managing Software Maniacs,* Wiley, 1994.
[ZAH94] Zahniser, R., "Timeboxing for Top Team Performance," *Software Development,* March 1994, pp. 35–38.

PROBLEMS AND POINTS TO PONDER

3.1. Based on information contained in this chapter and your own experience, develop "ten commandments" for empowering software engineers. That is, make a list of ten guidelines that will lead to software people who work to their full potential.

3.2. The Software Engineering Institute's people management capability maturity model (PM-CMM) takes an organized look at "key practice areas" that cultivate good software people. Your instructor will assign you one KPA for analysis and summary.

3.3. Describe three real-life situations in which the customer and the end-user are the same. Describe three situations in which they are different.

3.4. The decisions made by senior management can have a significant impact on the effectiveness of a software engineering team. Provide five examples to illustrate that this is true.

3.5. Review a copy of Weinberg's book [WEI86] and write a two- or three-page summary of the issues that should be considered in applying the MOI model.

3.6. You have been appointed a project manager within an information systems organization. Your job is to build an application that is quite similar to others your team has built, although this one is larger and more complex. Requirements have been thoroughly documented by the customer. What team structure would you choose and why? What software process model(s) would you choose and why?

3.7. You have been appointed a project manager for a small software products company. Your job is to build a breakthrough product that combines virtual reality hardware with state-of-the-art software. Because competition for the home entertainment market is intense, there is significant pressure to get the job done. What team struc-

ture would you choose and why? What software process model(s) would you choose and why?

3.8. You have been appointed a project manager for a major software products company. Your job is to manage the development of the next generation version of its widely used word-processing software. Because competition is intense, tight deadlines have been established and announced. What team structure would you choose and why? What software process model(s) would you choose and why?

3.9. You have been appointed a software project manager for a company that services the genetic engineering world. Your job is to manage the development of a new software product that will accelerate the pace of gene typing. The work is R&D oriented, but the goal to to produce a product within the next year. What team structure would you choose and why? What software process model(s) would you choose and why?

3.10. Referring to Figure 3.1, based on the results of the referenced study, documents are perceived to have more use than value. Why do you think this occurred and what can be done to move the documents data point above the regression line in the graph? That is, what can be done to improve the perceived value of documents?

3.11. You have been asked to develop a small application that analyzes each course offered by a university and reports the average grade obtained in the course (for a given term). Write a statement of scope that bounds this problem.

3.12. Do a first level functional decomposition of the page layout function discussed briefly in Section 3.3.2.

FURTHER READINGS AND INFORMATION SOURCES

An excellent four volume series written by Weinberg (*Quality Software Management,* Dorset House, 1992, 1993, 1994, 1996) introduces basic systems thinking and management concepts, explains how to use measurements effectively, and addresses "congruent action," the ability to establish "fit" between the manager's needs, the needs of technical staff, and the needs of the business. It will provide both new and experienced managers with useful information. Brooks (*The Mythical Man-Month,* Anniversary Edition, Addison-Wesley, 1995) has updated his classic book to provide new insight into software project and management issues. Purba and Shah (*How to Manage a Successful Software Project,* Wiley, 1995) present a number of case studies that indicate why some projects succeed and others fail. Bennatan (*Software Project Management in a Client/Server Environment,* Wiley, 1995) discusses special management issues associated with the development of client/server systems.

It can be argued that the most important aspect of software project management is people management. The definitive book on this subject has been written by

DeMarco and Lister [DEM98], but the following books on this subject have been published in recent years and are worth examining:

Beaudouin-Lafon, M., *Computer Supported Cooperative Work,* Wiley-Liss, 1999.

Carmel, E., *Global Software Teams: Collaborating Across Borders and Time Zones,* Prentice Hall, 1999.

Humphrey, W.S., *Managing Technical People: Innovation, Teamwork, and the Software Process,* Addison-Wesley, 1997.

Humphrey, W.S., *Introduction to the Team Software Process,* Addison-Wesley, 1999.

Jones, P.H., *Handbook of Team Design: A Practitioner's Guide to Team Systems Development,* McGraw-Hill, 1997.

Karolak, D.S., *Global Software Development: Managing Virtual Teams and Environments,* IEEE Computer Society, 1998.

Mayer, M., *The Virtual Edge: Embracing Technology for Distributed Project Team Success,* Project Management Institute Publications, 1999.

Another excellent book by Weinberg [WEI86] is must reading for every project manager and every team leader. It will give you insight and guidance in ways to do your job more effectively. House (*The Human Side of Project Management,* Addison-Wesley, 1988) and Crosby *(Running Things: The Art of Making Things Happen,* McGraw-Hill, 1989) provide practical advice for managers who must deal with human as well as technical problems.

Even though they do not relate specifically to the software world and sometimes suffer from over-simplification and broad generalization, best-selling "management" books by Drucker (*Management Challenges for the 21st Century,* Harper Business, 1999), Buckingham and Coffman (*First, Break All the Rules: What the World's Greatest Managers Do Differently,* Simon and Schuster, 1999) and Christensen (*The Innovator's Dilemma,* Harvard Business School Press, 1997) emphasize "new rules" defined by a rapidly changing economy, Older titles such as *The One-Minute Manager* and *In Search of Excellence* continue to provide valuable insights that can help you to manage people issues more effectively.

A wide variety of information sources on software project issues are available on the Internet. An up-to-date list of World Wide Web references that are relevant to the software projects can be found at the SEPA Web site:

http://www.mhhe.com/engcs/compsci/pressman/resources/project-mgmt.mhtml

CHAPTER

4 SOFTWARE PROCESS AND PROJECT METRICS

KEY CONCEPTS

backfiring 94
defect removal efficiency 98
function points . . . 89
metrics collection 100
project metrics . . . 86
process metrics . . 82
quality metrics . . . 95
size-oriented metrics 88
statistical process control 100
SSPI 84

Measurement is fundamental to any engineering discipline, and software engineering is no exception. Measurement enables us to gain insight by providing a mechanism for objective evaluation. Lord Kelvin once said:

> When you can measure what you are speaking about and express it in numbers, you know something about it; but when you cannot measure, when you cannot express it in numbers, your knowledge is of a meager and unsatisfactory kind: it may be the beginning of knowledge, but you have scarcely, in your thoughts, advanced to the stage of a science.

The software engineering community has finally begun to take Lord Kelvin's words to heart. But not without frustration and more than a little controversy!

Software metrics refers to a broad range of measurements for computer software. Measurement can be applied to the software process with the intent of improving it on a continuous basis. Measurement can be used throughout a software project to assist in estimation, quality control, productivity assessment, and project control. Finally, measurement can be used by software engineers to help assess the quality of technical work products and to assist in tactical decision making as a project proceeds.

QUICK LOOK

What is it? Software process and product metrics are quantitative measures that enable software people to gain insight into the efficacy of the software process and the projects that are conducted using the process as a framework. Basic quality and productivity data are collected. These data are then analyzed, compared against past averages, and assessed to determine whether quality and productivity improvements have occurred. Metrics are also used to pinpoint problem areas so that remedies can be developed and the software process can be improved.

Who does it? Software metrics are analyzed and assessed by software managers. Measures are often collected by software engineers.

Why is it important? If you don't measure, judgement can be based only on subjective evaluation. With measurement, trends (either good or bad) can be spotted, better estimates can be made, and true improvement can be accomplished over time.

What are the steps? Begin by defining a limited set of process, project, and product measures that are easy to collect. These measures are often normalized using either size- or function-oriented metrics. The result is analyzed and compared to past averages for similar projects performed within the organization. Trends are assessed and conclusions are generated.

QUICK LOOK

What is the work product? A set of software metrics that provide insight into the process and understanding of the project.

How do I ensure that I've done it right? By applying a consistent, yet simple measurement scheme that is never to be used to assess, reward, or punish individual performance.

XRef

Technical metrics for software engineering are presented in Chapters 19 and 24.

Within the context of software project management, we are concerned primarily with productivity and quality metrics—measures of software development "output" as a function of effort and time applied and measures of the "fitness for use" of the work products that are produced. For planning and estimating purposes, our interest is historical. What was software development productivity on past projects? What was the quality of the software that was produced? How can past productivity and quality data be extrapolated to the present? How can it help us plan and estimate more accurately?

In their guidebook on software measurement, Park, Goethert, and Florac [PAR96] discuss the reasons that we measure:

There are four reasons for measuring software processes, products, and resources: to characterize, to evaluate, to predict, or to improve.

We characterize to gain understanding of processes, products, resources, and environments, and to establish baselines for comparisons with future assessments.

We evaluate to determine status with respect to plans. Measures are the sensors that let us know when our projects and processes are drifting off track, so that we can bring them back under control. We also evaluate to assess achievement of quality goals and to assess the impacts of technology and process improvements on products and processes.

We predict so that we can plan. Measuring for prediction involves gaining understandings of relationships among processes and products and building models of these relationships, so that the values we observe for some attributes can be used to predict others. We do this because we want to establish achievable goals for cost, schedule, and quality—so that appropriate resources can be applied. Predictive measures are also the basis for extrapolating trends, so estimates for cost, time, and quality can be updated based on current evidence. Projections and estimates based on historical data also help us analyze risks and make design/cost trade-offs.

We measure to improve when we gather quantitative information to help us identify roadblocks, root causes, inefficiencies, and other opportunities for improving product quality and process performance.

Quote:

"Software metrics let you know when to laugh and when to cry."

Tom Gilb

4.1 MEASURES, METRICS, AND INDICATORS

Although the terms *measure, measurement,* and *metrics* are often used interchangeably, it is important to note the subtle differences between them. Because *measure*

can be used either as a noun or a verb, definitions of the term can become confusing. Within the software engineering context, a measure provides a quantitative indication of the extent, amount, dimension, capacity, or size of some attribute of a product or process. Measurement is the act of determining a measure. The *IEEE Standard Glossary of Software Engineering Terms* [IEE93] defines *metric* as "a quantitative measure of the degree to which a system, component, or process possesses a given attribute."

Quote:

"Not everything that can be counted counts, and not everything that counts can be counted."

Albert Einstein

When a single data point has been collected (e.g., the number of errors uncovered in the review of a single module), a measure has been established. Measurement occurs as the result of the collection of one or more data points (e.g., a number of module reviews are investigated to collect measures of the number of errors for each). A software metric relates the individual measures in some way (e.g., the average number of errors found per review or the average number of errors found per person-hour expended on reviews.[1]

A software engineer collects measures and develops metrics so that indicators will be obtained. An *indicator* is a metric or combination of metrics that provide insight into the software process, a software project, or the product itself [RAG95]. An indicator provides insight that enables the project manager or software engineers to adjust the process, the project, or the process to make things better.

For example, four software teams are working on a large software project. Each team must conduct design reviews but is allowed to select the type of review that it will use. Upon examination of the metric, errors found per person-hour expended, the project manager notices that the two teams using more formal review methods exhibit an errors found per person-hour expended that is 40 percent higher than the other teams. Assuming all other parameters equal, this provides the project manager with an indicator that formal review methods may provide a higher return on time investment than another, less formal review approach. She may decide to suggest that all teams use the more formal approach. The metric provides the manager with insight. And insight leads to informed decision making.

4.2 METRICS IN THE PROCESS AND PROJECT DOMAINS

Measurement is commonplace in the engineering world. We measure power consumption, weight, physical dimensions, temperature, voltage, signal-to-noise ratio . . . the list is almost endless. Unfortunately, measurement is far less common in the software engineering world. We have trouble agreeing on what to measure and trouble evaluating measures that are collected.

1 This assumes that another measure, person-hours expended, is collected for each review.

A comprehensive software metrics guidebook can be downloaded from **www.ivv.nasa.gov/SWG/resources/NASA-GB-001-94.pdf**

Metrics should be collected so that process and product indicators can be ascertained. *Process indicators* enable a software engineering organization to gain insight into the efficacy of an existing process (i.e., the paradigm, software engineering tasks, work products, and milestones). They enable managers and practitioners to assess what works and what doesn't. Process metrics are collected across all projects and over long periods of time. Their intent is to provide indicators that lead to long-term software process improvement.

Project indicators enable a software project manager to (1) assess the status of an ongoing project, (2) track potential risks, (3) uncover problem areas before they go "critical," (4) adjust work flow or tasks, and (5) evaluate the project team's ability to control quality of software work products.

In some cases, the same software metrics can be used to determine project and then process indicators. In fact, measures that are collected by a project team and converted into metrics for use during a project can also be transmitted to those with responsibility for software process improvement. For this reason, many of the same metrics are used in both the process and project domain.

4.2.1 Process Metrics and Software Process Improvement

The only rational way to improve any process is to measure specific attributes of the process, develop a set of meaningful metrics based on these attributes, and then use the metrics to provide indicators that will lead to a strategy for improvement. But before we discuss software metrics and their impact on software process improvement, it is important to note that process is only one of a number of "controllable factors in improving software quality and organizational performance [PAU94]."

The skill and motivation of the people doing the work are the most important factors that influence software quality.

Referring to Figure 4.1, process sits at the center of a triangle connecting three factors that have a profound influence on software quality and organizational performance. The skill and motivation of people has been shown [BOE81] to be the single most influential factor in quality and performance. The complexity of the product can have a substantial impact on quality and team performance. The technology (i.e., the software engineering methods) that populate the process also has an impact. In addition, the process triangle exists within a circle of environmental conditions that include the development environment (e.g., CASE tools), business conditions (e.g., deadlines, business rules), and customer characteristics (e.g., ease of communication).

How do I measure the effectiveness of a software process?

We measure the efficacy of a software process indirectly. That is, we derive a set of metrics based on the outcomes that can be derived from the process. Outcomes include measures of errors uncovered before release of the software, defects delivered to and reported by end-users, work products delivered (productivity), human effort expended, calendar time expended, schedule conformance, and other measures. We also derive process metrics by measuring the characteristics of specific software engineering tasks. For example, we might measure the effort and time spent

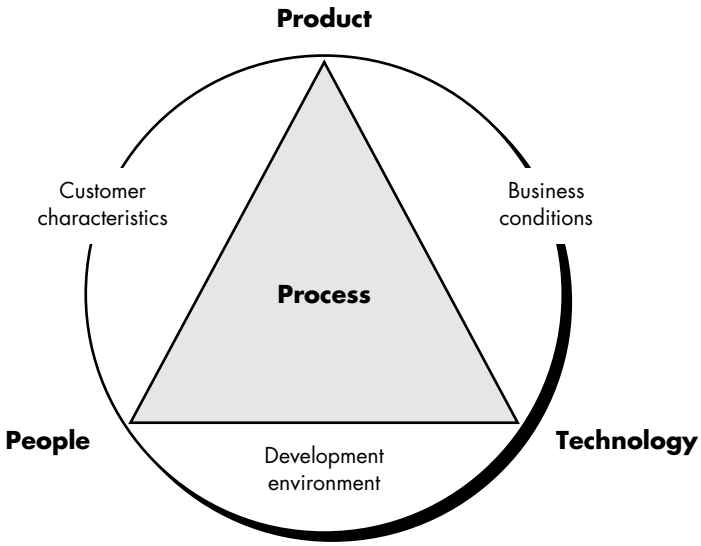


FIGURE 4.1 Determinants for software quality and organizational effectiveness (adapted from [PAU94])

performing the umbrella activities and the generic software engineering activities described in Chapter 2.

Grady [GRA92] argues that there are "private and public" uses for different types of process data. Because it is natural that individual software engineers might be sensitive to the use of metrics collected on an individual basis, these data should be private to the individual and serve as an indicator for the individual only. Examples of *private metrics* include defect rates (by individual), defect rates (by module), and errors found during development.

The "private process data" philosophy conforms well with the *personal software process* approach proposed by Humphrey [HUM95]. Humphrey describes the approach in the following manner:

> The personal software process (PSP) is a structured set of process descriptions, measurements, and methods that can help engineers to improve their personal performance. It provides the forms, scripts, and standards that help them estimate and plan their work. It shows them how to define processes and how to measure their quality and productivity. A fundamental PSP principle is that everyone is different and that a method that is effective for one engineer may not be suitable for another. The PSP thus helps engineers to measure and track their own work so they can find the methods that are best for them.

Humphrey recognizes that software process improvement can and should begin at the individual level. Private process data can serve as an important driver as the individual software engineer works to improve.

Some process metrics are private to the software project team but public to all team members. Examples include defects reported for major software functions (that

have been developed by a number of practitioners), errors found during formal technical reviews, and lines of code or function points per module and function.[2] These data are reviewed by the team to uncover indicators that can improve team performance.

Public metrics generally assimilate information that originally was private to individuals and teams. Project level defect rates (absolutely not attributed to an individual), effort, calendar times, and related data are collected and evaluated in an attempt to uncover indicators that can improve organizational process performance.

KEY POINT

Public metrics enable an organization to make strategic changes that improve the software process and tactical changes during a software project.

Software process metrics can provide significant benefit as an organization works to improve its overall level of process maturity. However, like all metrics, these can be misused, creating more problems than they solve. Grady [GRA92] suggests a "software metrics etiquette" that is appropriate for both managers and practitioners as they institute a process metrics program:

- Use common sense and organizational sensitivity when interpreting metrics data.
- Provide regular feedback to the individuals and teams who collect measures and metrics.
- Don't use metrics to appraise individuals.
- Work with practitioners and teams to set clear goals and metrics that will be used to achieve them.
- Never use metrics to threaten individuals or teams.
- Metrics data that indicate a problem area should not be considered "negative." These data are merely an indicator for process improvement.
- Don't obsess on a single metric to the exclusion of other important metrics.

? What guidelines should be applied when we collect software metrics?

As an organization becomes more comfortable with the collection and use of process metrics, the derivation of simple indicators gives way to a more rigorous approach called *statistical software process improvement* (SSPI). In essence, SSPI uses software failure analysis to collect information about all errors and defects[3] encountered as an application, system, or product is developed and used. Failure analysis works in the following manner:

1. All errors and defects are categorized by origin (e.g., flaw in specification, flaw in logic, nonconformance to standards).
2. The cost to correct each error and defect is recorded.

WebRef

SSPI and other quality related information is available through the American Society for Quality at **www.asq.org**

2 See Sections 4.3.1 and 4.3.2 for detailed discussions of LOC and function point metrics.

3 As we discuss in Chapter 8, an *error* is some flaw in a software engineering work product or deliverable that is uncovered by software engineers before the software is delivered to the end-user. A *defect* is a flaw that is uncovered after delivery to the end-user.

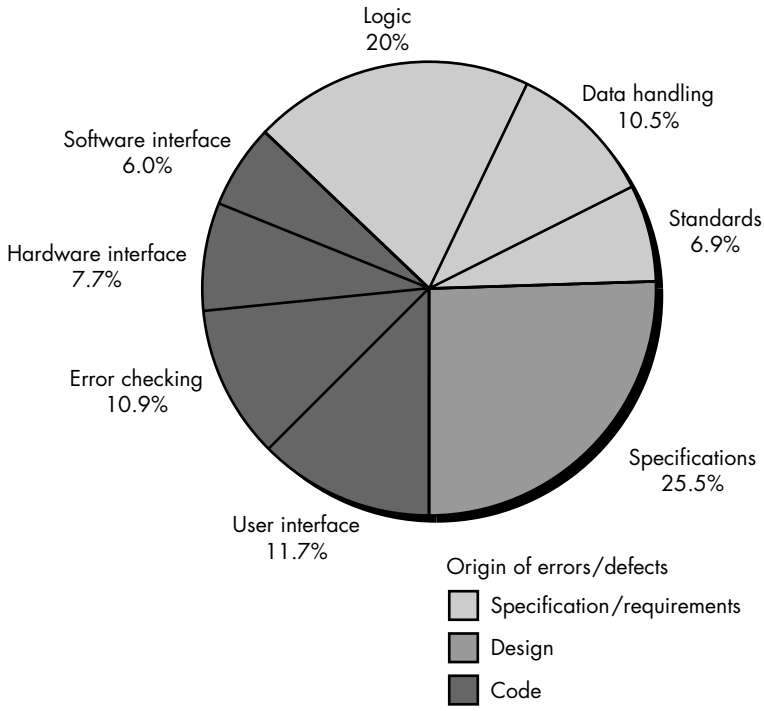


FIGURE 4.2
Causes of defects and their origin for four software projects [GRA94]

You can't improve your approach to software engineering unless you understand where you're strong and where you're weak. Use SSPI techniques to gain that understanding.

3. The number of errors and defects in each category is counted and ranked in descending order.
4. The overall cost of errors and defects in each category is computed.
5. Resultant data are analyzed to uncover the categories that result in highest cost to the organization.
6. Plans are developed to modify the process with the intent of eliminating (or reducing the frequency of) the class of errors and defects that is most costly.

Following steps 1 and 2, a simple defect distribution can be developed (Figure 4.2) [GRA94]. For the pie-chart noted in the figure, eight causes of defects and their origin (indicated by shading) are shown. Grady suggests the development of a *fishbone diagram* [GRA92] to help in diagnosing the data represented in the frequency diagram. Referring to Figure 4.3, the spine of the diagram (the central line) represents the quality factor under consideration (in this case specification defects that account for 25 percent of the total). Each of the ribs (diagonal lines) connecting to the spine indicate potential causes for the quality problem (e.g., missing requirements, ambiguous specification, incorrect requirements, changed requirements). The spine and ribs notation is then added to each of the major ribs of the diagram to expand upon the cause noted. Expansion is shown only for the *incorrect* cause in Figure 4.3.

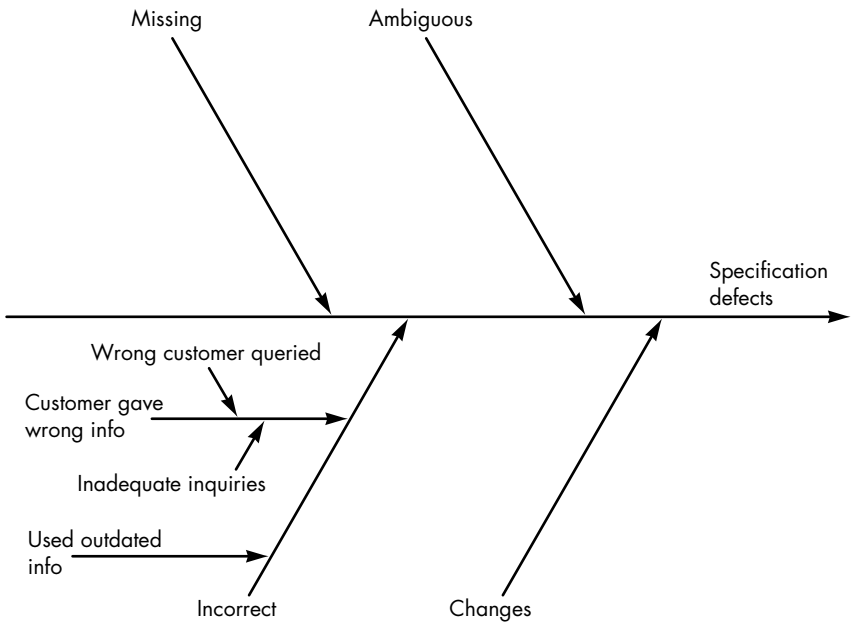


FIGURE 4.3 A fishbone diagram (adapted from [GRA92])

The collection of process metrics is the driver for the creation of the fishbone diagram. A completed fishbone diagram can be analyzed to derive indicators that will enable a software organization to modify its process to reduce the frequency of errors and defects.

4.2.2 Project Metrics

Software process metrics are used for strategic purposes. Software project measures are tactical. That is, project metrics and the indicators derived from them are used by a project manager and a software team to adapt project work flow and technical activities.

XRef

Project estimation techniques are discussed in Chapter 5.

The first application of project metrics on most software projects occurs during estimation. Metrics collected from past projects are used as a basis from which effort and time estimates are made for current software work. As a project proceeds, measures of effort and calendar time expended are compared to original estimates (and the project schedule). The project manager uses these data to monitor and control progress.

As technical work commences, other project metrics begin to have significance. Production rates represented in terms of pages of documentation, review hours, function points, and delivered source lines are measured. In addition, errors uncovered during each software engineering task are tracked. As the software evolves from specification into design, technical metrics (Chapters 19 and 24) are collected to assess

How should we use metrics during the project itself?

design quality and to provide indicators that will influence the approach taken to code generation and testing.

The intent of project metrics is twofold. First, these metrics are used to minimize the development schedule by making the adjustments necessary to avoid delays and mitigate potential problems and risks. Second, project metrics are used to assess product quality on an ongoing basis and, when necessary, modify the technical approach to improve quality.

As quality improves, defects are minimized, and as the defect count goes down, the amount of rework required during the project is also reduced. This leads to a reduction in overall project cost.

Another model of software project metrics [HET93] suggests that every project should measure:

- *Inputs*—measures of the resources (e.g., people, environment) required to do the work.
- *Outputs*—measures of the deliverables or work products created during the software engineering process.
- *Results*—measures that indicate the effectiveness of the deliverables.

In actuality, this model can be applied to both process and project. In the project context, the model can be applied recursively as each framework activity occurs. Therefore the output from one activity becomes input to the next. Results metrics can be used to provide an indication of the usefulness of work products as they flow from one framework activity (or task) to the next.

4.3 SOFTWARE MEASUREMENT

Measurements in the physical world can be categorized in two ways: direct measures (e.g., the length of a bolt) and indirect measures (e.g., the "quality" of bolts produced, measured by counting rejects). Software metrics can be categorized similarly.

What is the difference between direct and indirect measures?

Direct measures of the software engineering process include cost and effort applied. Direct measures of the product include lines of code (LOC) produced, execution speed, memory size, and defects reported over some set period of time. *Indirect measures* of the product include functionality, quality, complexity, efficiency, reliability, maintainability, and many other "–abilities" that are discussed in Chapter 19.

The cost and effort required to build software, the number of lines of code produced, and other direct measures are relatively easy to collect, as long as specific conventions for measurement are established in advance. However, the quality and functionality of software or its efficiency or maintainability are more difficult to assess and can be measured only indirectly.

We have already partitioned the software metrics domain into process, project, and product metrics. We have also noted that product metrics that are private to an

individual are often combined to develop project metrics that are public to a software team. Project metrics are then consolidated to create process metrics that are public to the software organization as a whole. But how does an organization combine metrics that come from different individuals or projects?

Because many factors influence software work, don't use metrics to compare individuals or teams.

To illustrate, we consider a simple example. Individuals on two different project teams record and categorize all errors that they find during the software process. Individual measures are then combined to develop team measures. Team A found 342 errors during the software process prior to release. Team B found 184 errors. All other things being equal, which team is more effective in uncovering errors throughout the process? Because we do not know the size or complexity of the projects, we cannot answer this question. However, if the measures are normalized, it is possible to create software metrics that enable comparison to broader organizational averages.

4.3.1 Size-Oriented Metrics

What data should we collect to derive size-oriented metrics?

Size-oriented software metrics are derived by normalizing quality and/or productivity measures by considering the *size* of the software that has been produced. If a software organization maintains simple records, a table of size-oriented measures, such as the one shown in Figure 4.4, can be created. The table lists each software development project that has been completed over the past few years and corresponding measures for that project. Referring to the table entry (Figure 4.4) for project alpha: 12,100 lines of code were developed with 24 person-months of effort at a cost of $168,000. It should be noted that the effort and cost recorded in the table represent all software engineering activities (analysis, design, code, and test), not just coding. Further information for project alpha indicates that 365 pages of documentation were developed, 134 errors were recorded before the software was released, and 29 defects

Project	LOC	Effort	$(000)	Pp. doc.	Errors	Defects	People
alpha	12,100	24	168	365	134	29	3
beta	27,200	62	440	1224	321	86	5
gamma	20,200	43	314	1050	256	64	6
•	•	•	•	•	•		
•	•	•	•	•	•		
•	•	•	•	•	•		

FIGURE 4.4 Size-oriented metrics

were encountered after release to the customer within the first year of operation. Three people worked on the development of software for project alpha.

Metrics collection template

In order to develop metrics that can be assimilated with similar metrics from other projects, we choose lines of code as our normalization value. From the rudimentary data contained in the table, a set of simple size-oriented metrics can be developed for each project:

- Errors per KLOC (thousand lines of code).
- Defects[4] per KLOC.
- $ per LOC.
- Page of documentation per KLOC.

In addition, other interesting metrics can be computed:

- Errors per person-month.
- LOC per person-month.
- $ per page of documentation.

Size-oriented metrics are not universally accepted as the best way to measure the process of software development [JON86]. Most of the controversy swirls around the use of lines of code as a key measure. Proponents of the LOC measure claim that LOC is an "artifact" of all software development projects that can be easily counted, that many existing software estimation models use LOC or KLOC as a key input, and that a large body of literature and data predicated on LOC already exists. On the other hand, opponents argue that LOC measures are programming language dependent, that they penalize well-designed but shorter programs, that they cannot easily accommodate nonprocedural languages, and that their use in estimation requires a level of detail that may be difficult to achieve (i.e., the planner must estimate the LOC to be produced long before analysis and design have been completed).

Size-oriented metrics are widely used, but debate about their validity and applicability continues.

4.3.2 Function-Oriented Metrics

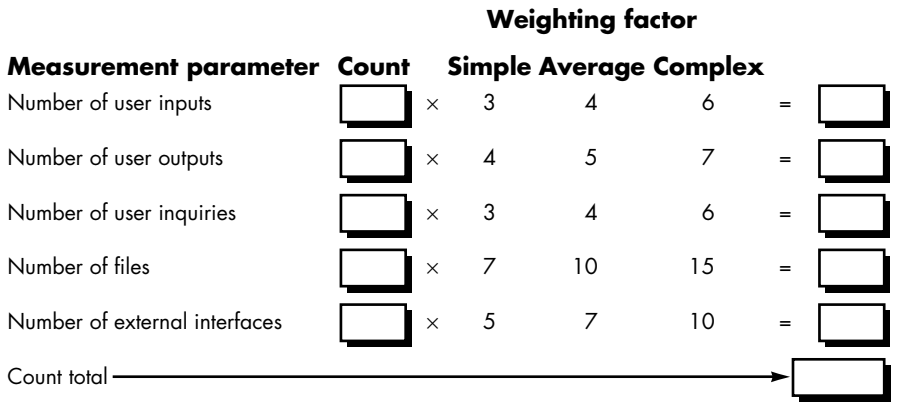
Function-oriented software metrics use a measure of the functionality delivered by the application as a normalization value. Since 'functionality' cannot be measured directly, it must be derived indirectly using other direct measures. Function-oriented metrics were first proposed by Albrecht [ALB79], who suggested a measure called the *function point.* Function points are derived using an empirical relationship based on countable (direct) measures of software's information domain and assessments of software complexity.

Comprehensive information on function points can be obtained at **www.ifpug.org**

Function points are computed [IFP94] by completing the table shown in Figure 4.5. Five information domain characteristics are determined and counts are provided in

4 A defect occurs when quality assurance activities (e.g., formal technical reviews) fail to uncover an error in a work product produced during the software process.

FIGURE 4.5 Computing function points



the appropriate table location. Information domain values are defined in the following manner:[5]

Number of user inputs. Each user input that provides distinct application-oriented data to the software is counted. Inputs should be distinguished from inquiries, which are counted separately.

Number of user outputs. Each user output that provides application-oriented information to the user is counted. In this context output refers to reports, screens, error messages, etc. Individual data items within a report are not counted separately.

Number of user inquiries. An inquiry is defined as an on-line input that results in the generation of some immediate software response in the form of an on-line output. Each distinct inquiry is counted.

Number of files. Each logical master file (i.e., a logical grouping of data that may be one part of a large database or a separate file) is counted.

Number of external interfaces. All machine readable interfaces (e.g., data files on storage media) that are used to transmit information to another system are counted.

KEY POINT

Function points are derived from direct measures of the information domain.

Once these data have been collected, a complexity value is associated with each count. Organizations that use function point methods develop criteria for determining whether a particular entry is simple, average, or complex. Nonetheless, the determination of complexity is somewhat subjective.

To compute function points (FP), the following relationship is used:

$$\text{FP} = \text{count total} \times [0.65 + 0.01 \times \Sigma(F_i)] \qquad (4\text{-}1)$$

where count total is the sum of all FP entries obtained from Figure 4.5.

5 In actuality, the definition of information domain values and the manner in which they are counted are a bit more complex. The interested reader should see [IFP94] for details.

The F_i (i = 1 to 14) are "complexity adjustment values" based on responses to the following questions [ART85]:

1. Does the system require reliable backup and recovery?
2. Are data communications required?
3. Are there distributed processing functions?
4. Is performance critical?
5. Will the system run in an existing, heavily utilized operational environment?
6. Does the system require on-line data entry?
7. Does the on-line data entry require the input transaction to be built over multiple screens or operations?
8. Are the master files updated on-line?
9. Are the inputs, outputs, files, or inquiries complex?
10. Is the internal processing complex?
11. Is the code designed to be reusable?
12. Are conversion and installation included in the design?
13. Is the system designed for multiple installations in different organizations?
14. Is the application designed to facilitate change and ease of use by the user?

Each of these questions is answered using a scale that ranges from 0 (not important or applicable) to 5 (absolutely essential). The constant values in Equation (4-1) and the weighting factors that are applied to information domain counts are determined empirically.

Once function points have been calculated, they are used in a manner analogous to LOC as a way to normalize measures for software productivity, quality, and other attributes:

- Errors per FP.
- Defects per FP.
- $ per FP.
- Pages of documentation per FP.
- FP per person-month.

4.3.3 Extended Function Point Metrics

KEY POINT

Extending function points are used for engineering, real-time, and control-oriented applications.

The function point measure was originally designed to be applied to business information systems applications. To accommodate these applications, the data dimension (the information domain values discussed previously) was emphasized to the exclusion of the functional and behavioral (control) dimensions. For this reason, the function point measure was inadequate for many engineering and embedded systems (which emphasize function and control). A number of extensions to the basic function point measure have been proposed to remedy this situation.

A function point extension called *feature points* [JON91], is a superset of the function point measure that can be applied to systems and engineering software applications.

The feature point measure accommodates applications in which algorithmic complexity is high. Real-time, process control and embedded software applications tend to have high algorithmic complexity and are therefore amenable to the feature point.

To compute the feature point, information domain values are again counted and weighted as described in Section 4.3.2. In addition, the feature point metric counts a new software characteristic—*algorithms*. An *algorithm* is defined as "a bounded computational problem that is included within a specific computer program" [JON91]. Inverting a matrix, decoding a bit string, or handling an interrupt are all examples of algorithms.

A useful FAQ on function points (and extended function points) can be obtained at **http://ourworld.compuserve.com/homepages/softcomp/**

Another function point extension for real-time systems and engineered products has been developed by Boeing. The Boeing approach integrates the data dimension of software with the functional and control dimensions to provide a function-oriented measure amenable to applications that emphasize function and control capabilities. Called the *3D function point* [WHI95], characteristics of all three software dimensions are "counted, quantified, and transformed" into a measure that provides an indication of the functionality delivered by the software.[6]

The *data dimension* is evaluated in much the same way as described in Section 4.3.2. Counts of retained data (the internal program data structure; e.g., files) and external data (inputs, outputs, inquiries, and external references) are used along with measures of complexity to derive a data dimension count. The *functional dimension* is measured by considering "the number of internal operations required to transform input to output data" [WHI95]. For the purposes of 3D function point computation, a "transformation" is viewed as a series of processing steps that are constrained by a set of semantic statements. The *control dimension* is measured by counting the number of transitions between states.[7]

A state represents some externally observable mode of behavior, and a transition occurs as a result of some event that causes the software or system to change its mode of behavior (i.e., to change state). For example, a wireless phone contains software that supports auto dial functions. To enter the *auto-dial* state from a *resting state,* the user presses an **Auto** key on the keypad. This event causes an LCD display to prompt for a code that will indicate the party to be called. Upon entry of the code and hitting the **Dial** key (another event), the wireless phone software makes a transition to the *dialing* state. When computing 3D function points, transitions are not assigned a complexity value.

To compute 3D function points, the following relationship is used:

$$\text{index} = I + O + Q + F + E + T + R \qquad (4\text{-}2)$$

---

6 It should be noted that other extensions to function points for application in real-time software work (e.g., [ALA97]) have also been proposed. However, none of these appears to be widely used in the industry.

7 A detailed discussion of the behavioral dimension, including states and state transitions, is presented in Chapter 12.

FIGURE 4.6
Determining the complexity of a transformation for 3D function points [WHI95].

Semantic statements / Processing steps	1–5	6–10	11+
1–10	Low	Low	Average
11–20	Low	Average	High
21+	Average	High	High

where *I, O, Q, F, E, T*, and *R* represent complexity weighted values for the elements discussed already: inputs, outputs, inquiries, internal data structures, external files, transformation, and transitions, respectively. Each complexity weighted value is computed using the following relationship:

$$\text{complexity weighted value} = N_{il}W_{il} + N_{ia}W_{ia} + N_{ih}W_{ih} \quad (4\text{-}3)$$

where N_{il}, N_{ia}, and N_{ih} represent the number of occurrences of element i (e.g., outputs) for each level of complexity (low, medium, high); and W_{il}, W_{ia}, and W_{ih} are the corresponding weights. The overall complexity of a transformation for 3D function points is shown in Figure 4.6.

It should be noted that function points, feature points, and 3D function points represent the same thing—"functionality" or "utility" delivered by software. In fact, each of these measures results in the same value if only the data dimension of an application is considered. For more complex real-time systems, the feature point count is often between 20 and 35 percent higher than the count determined using function points alone.

The function point (and its extensions), like the LOC measure, is controversial. Proponents claim that FP is programming language independent, making it ideal for applications using conventional and nonprocedural languages; that it is based on data that are more likely to be known early in the evolution of a project, making FP more attractive as an estimation approach. Opponents claim that the method requires some "sleight of hand" in that computation is based on subjective rather than objective data; that counts of the information domain (and other dimensions) can be difficult to collect after the fact; and that FP has no direct physical meaning—it's just a number.

4.4 RECONCILING DIFFERENT METRICS APPROACHES

The relationship between lines of code and function points depends upon the programming language that is used to implement the software and the quality of the design. A number of studies have attempted to relate FP and LOC measures. To quote Albrecht and Gaffney [ALB83]:

> The thesis of this work is that the amount of function to be provided by the application (program) can be estimated from the itemization of the major components[8] of data to be used or provided by it. Furthermore, this estimate of function should be correlated to both the amount of LOC to be developed and the development effort needed.

The following table [JON98] provides rough estimates of the average number of lines of code required to build one function point in various programming languages:

If I know the number of LOC, is it possible to estimate the number of function points?

Programming Language	LOC/FP (average)
Assembly language	320
C	128
COBOL	106
FORTRAN	106
Pascal	90
C++	64
Ada95	53
Visual Basic	32
Smalltalk	22
Powerbuilder (code generator)	16
SQL	12

Use backfiring data judiciously. It is far better to compute FP using the methods discussed earlier.

A review of these data indicates that one LOC of C++ provides approximately 1.6 times the "functionality" (on average) as one LOC of FORTRAN. Furthermore, one LOC of a Visual Basic provides more than three times the functionality of a LOC for a conventional programming language. More detailed data on the relationship between FP and LOC are presented in [JON98] and can be used to "backfire" (i.e., to compute the number of function points when the number of delivered LOC are known) existing programs to determine the FP measure for each.

LOC and FP measures are often used to derive productivity metrics. This invariably leads to a debate about the use of such data. Should the LOC/person-month (or FP/person-month) of one group be compared to similar data from another? Should managers appraise the performance of individuals by using these metrics? The answers

8 It is important to note that "the itemization of major components" can be interpreted in a variety of ways. Some software engineers who work in an object-oriented development environment (Part Four) use the number of classes or objects as the dominant size metric. A maintenance organization might view project size in terms of the number of engineering change orders (Chapter 9). An information systems organization might view the number of business processes affected by an application.

to these questions is an emphatic "No!" The reason for this response is that many factors influence productivity, making for "apples and oranges" comparisons that can be easily misinterpreted.

Function points and LOC based metrics have been found to be relatively accurate predictors of software development effort and cost. However, in order to use LOC and FP for estimation (Chapter 5), a historical baseline of information must be established.

4.5 METRICS FOR SOFTWARE QUALITY

The overriding goal of software engineering is to produce a high-quality system, application, or product. To achieve this goal, software engineers must apply effective methods coupled with modern tools within the context of a mature software process. In addition, a good software engineer (and good software engineering managers) must measure if high quality is to be realized.

An excellent source of information on software quality and related topics (including metrics) can be found at **www.qualityworld.com**

The quality of a system, application, or product is only as good as the requirements that describe the problem, the design that models the solution, the code that leads to an executable program, and the tests that exercise the software to uncover errors. A good software engineer uses measurement to assess the quality of the analysis and design models, the source code, and the test cases that have been created as the software is engineered. To accomplish this real-time quality assessment, the engineer must use technical measures (Chapters 19 and 24) to evaluate quality in objective, rather than subjective ways.

XRef

A detailed discussion of software quality assurance activities is presented in Chapter 8.

The project manager must also evaluate quality as the project progresses. Private metrics collected by individual software engineers are assimilated to provide project-level results. Although many quality measures can be collected, the primary thrust at the project level is to measure errors and defects. Metrics derived from these measures provide an indication of the effectiveness of individual and group software quality assurance and control activities.

Metrics such as work product (e.g., requirements or design) errors per function point, errors uncovered per review hour, and errors uncovered per testing hour provide insight into the efficacy of each of the activities implied by the metric. Error data can also be used to compute the *defect removal efficiency* (DRE) for each process framework activity. DRE is discussed in Section 4.5.3.

4.5.1 An Overview of Factors That Affect Quality

Over 25 years ago, McCall and Cavano [MCC78] defined a set of quality factors that were a first step toward the development of metrics for software quality. These factors assess software from three distinct points of view: (1) product operation (using it), (2) product revision (changing it), and (3) product transition (modifying it to work in a different environment; i.e., "porting" it). In their work, the authors describe the

relationship between these quality factors (what they call a *framework*) and other aspects of the software engineering process:

> First, the framework provides a mechanism for the project manager to identify what qualities are important. These qualities are attributes of the software in addition to its functional correctness and performance which have life cycle implications. Such factors as maintainability and portability have been shown in recent years to have significant life cycle cost impact . . .
>
> Secondly, the framework provides a means for quantitatively assessing how well the development is progressing relative to the quality goals established . . .
>
> Thirdly, the framework provides for more interaction of QA personnel throughout the development effort . . .
>
> Lastly, . . . quality assurance personal can use indications of poor quality to help identify [better] standards to be enforced in the future.

Surprisingly, the factors that defined software quality in the 1970s are the same factors that continue to define software quality in the first decade of this century.

A detailed discussion of McCall and Cavano's framework, as well as other quality factors, is presented in Chapter 19. It is interesting to note that nearly every aspect of computing has undergone radical change as the years have passed since McCall and Cavano did their seminal work in 1978. But the attributes that provide an indication of software quality remain the same.

What does this mean? If a software organization adopts a set of quality factors as a "checklist" for assessing software quality, it is likely that software built today will still exhibit quality well into the first few decades of this century. Even as computing architectures undergo radical change (as they surely will), software that exhibits high quality in operation, transition, and revision will continue to serve its users well.

4.5.2 Measuring Quality

Although there are many measures of software quality, correctness, maintainability, integrity, and usability provide useful indicators for the project team. Gilb [GIL88] suggests definitions and measures for each.

Correctness. A program must operate correctly or it provides little value to its users. Correctness is the degree to which the software performs its required function. The most common measure for correctness is defects per KLOC, where a defect is defined as a verified lack of conformance to requirements. When considering the overall quality of a software product, defects are those problems reported by a user of the program after the program has been released for general use. For quality assessment purposes, defects are counted over a standard period of time, typically one year.

Maintainability. Software maintenance accounts for more effort than any other software engineering activity. Maintainability is the ease with which a program can be corrected if an error is encountered, adapted if its environment changes, or enhanced if the customer desires a change in require-

ments. There is no way to measure maintainability directly; therefore, we must use indirect measures. A simple time-oriented metric is *mean-time-to-change* (MTTC), the time it takes to analyze the change request, design an appropriate modification, implement the change, test it, and distribute the change to all users. On average, programs that are maintainable will have a lower MTTC (for equivalent types of changes) than programs that are not maintainable.

Hitachi [TAJ81] has used a cost-oriented metric for maintainability called *spoilage*—the cost to correct defects encountered after the software has been released to its end-users. When the ratio of spoilage to overall project cost (for many projects) is plotted as a function of time, a manager can determine whether the overall maintainability of software produced by a software development organization is improving. Actions can then be taken in response to the insight gained from this information.

Integrity. Software integrity has become increasingly important in the age of hackers and firewalls. This attribute measures a system's ability to withstand attacks (both accidental and intentional) to its security. Attacks can be made on all three components of software: programs, data, and documents.

To measure integrity, two additional attributes must be defined: threat and security. *Threat* is the probability (which can be estimated or derived from empirical evidence) that an attack of a specific type will occur within a given time. *Security* is the probability (which can be estimated or derived from empirical evidence) that the attack of a specific type will be repelled. The integrity of a system can then be defined as

$$\text{integrity} = \text{summation } [(1 - \text{threat}) \times (1 - \text{security})]$$

where threat and security are summed over each type of attack.

Usability. The catch phrase "user-friendliness" has become ubiquitous in discussions of software products. If a program is not user-friendly, it is often doomed to failure, even if the functions that it performs are valuable. Usability is an attempt to quantify user-friendliness and can be measured in terms of four characteristics: (1) the physical and or intellectual skill required to learn the system, (2) the time required to become moderately efficient in the use of the system, (3) the net increase in productivity (over the approach that the system replaces) measured when the system is used by someone who is moderately efficient, and (4) a subjective assessment (sometimes obtained through a questionnaire) of users attitudes toward the system. Detailed discussion of this topic is contained in Chapter 15.

The four factors just described are only a sampling of those that have been proposed as measures for software quality. Chapter 19 considers this topic in additional detail.

4.5.3 Defect Removal Efficiency

A quality metric that provides benefit at both the project and process level is defect removal efficiency (DRE). In essence, DRE is a measure of the filtering ability of quality assurance and control activities as they are applied throughout all process framework activities.

When considered for a project as a whole, DRE is defined in the following manner:

What is defect removal efficiency?

$$\text{DRE} = E/(E + D) \tag{4-4}$$

where E is the number of errors found before delivery of the software to the end-user and D is the number of defects found after delivery.

The ideal value for DRE is 1. That is, no defects are found in the software. Realistically, D will be greater than 0, but the value of DRE can still approach 1. As E increases (for a given value of D), the overall value of DRE begins to approach 1. In fact, as E increases, it is likely that the final value of D will decrease (errors are filtered out before they become defects). If used as a metric that provides an indicator of the filtering ability of quality control and assurance activities, DRE encourages a software project team to institute techniques for finding as many errors as possible before delivery.

DRE can also be used within the project to assess a team's ability to find errors before they are passed to the next framework activity or software engineering task. For example, the requirements analysis task produces an analysis model that can be reviewed to find and correct errors. Those errors that are not found during the review of the analysis model are passed on to the design task (where they may or may not be found). When used in this context, we redefine DRE as

ADVICE

Use DRE as a measure of the efficacy of your early SQA activities. If DRE is low during analysis and design, spend some time improving the way you conduct formal technical reviews.

$$\text{DRE}_i = E_i/(E_i + E_{i+1}) \tag{4-5}$$

where E_i is the number of errors found during software engineering activity i and E_{i+1} is the number of errors found during software engineering activity $i+1$ that are traceable to errors that were not discovered in software engineering activity i.

A quality objective for a software team (or an individual software engineer) is to achieve DRE_i that approaches 1. That is, errors should be filtered out before they are passed on to the next activity.

4.6 INTEGRATING METRICS WITHIN THE SOFTWARE PROCESS

The majority of software developers still do not measure, and sadly, most have little desire to begin. As we noted earlier in this chapter, the problem is cultural. Attempting to collect measures where none had been collected in the past often precipitates resistance. "Why do we need to do this?" asks a harried project manager. "I don't see the point," complains an overworked practitioner.

In this section, we consider some arguments for software metrics and present an approach for instituting a metrics collection program within a software engineering

organization. But before we begin, some words of wisdom are suggested by Grady and Caswell [GRA87]:

> Some of the things we describe here will sound quite easy. Realistically, though, establishing a successful company-wide software metrics program is hard work. When we say that you must wait at least three years before broad organizational trends are available, you get some idea of the scope of such an effort.

The caveat suggested by the authors is well worth heeding, but the benefits of measurement are so compelling that the hard work is worth it.

4.6.1 Arguments for Software Metrics

Why is it so important to measure the process of software engineering and the product (software) that it produces? The answer is relatively obvious. If we do not measure, there no real way of determining whether we are improving. And if we are not improving, we are lost.

By requesting and evaluating productivity and quality measures, senior management can establish meaningful goals for improvement of the software engineering process. In Chapter 1 we noted that software is a strategic business issue for many companies. If the process through which it is developed can be improved, a direct impact on the bottom line can result. But to establish goals for improvement, the current status of software development must be understood. Hence, measurement is used to establish a process baseline from which improvements can be assessed.

Quote:

"We manage things 'by the numbers' in many aspects of our lives. . . . These numbers give us insight and help steer our actions."

Michael Mah
Larry Putnam

The day-to-day rigors of software project work leave little time for strategic thinking. Software project managers are concerned with more mundane (but equally important) issues: developing meaningful project estimates, producing higher-quality systems, getting product out the door on time. By using measurement to establish a project baseline, each of these issues becomes more manageable. We have already noted that the baseline serves as a basis for estimation. Additionally, the collection of quality metrics enables an organization to "tune" its software process to remove the "vital few" causes of defects that have the greatest impact on software development.[9]

At the project and technical levels (in the trenches), software metrics provide immediate benefit. As the software design is completed, most developers would be anxious to obtain answers to the questions such as

- Which user requirements are most likely to change?
- Which components in this system are most error prone?
- How much testing should be planned for each component?
- How many errors (of specific types) can I expect when testing commences?

---

9 These ideas have been formalized into an approach called *statistical software quality assurance* and are discussed in detail in Chapter 8.

Answers to these questions can be determined if metrics have been collected and used as a technical guide. In later chapters, we examine how this is done.

4.6.2 Establishing a Baseline

By establishing a metrics baseline, benefits can be obtained at the process, project, and product (technical) levels. Yet the information that is collected need not be fundamentally different. The same metrics can serve many masters. The metrics baseline consists of data collected from past software development projects and can be as simple as the table presented in Figure 4.4 or as complex as a comprehensive database containing dozens of project measures and the metrics derived from them.

What critical information can metrics provide for a developer?

To be an effective aid in process improvement and/or cost and effort estimation, baseline data must have the following attributes: (1) data must be reasonably accurate—"guestimates" about past projects are to be avoided; (2) data should be collected for as many projects as possible; (3) measures must be consistent, for example, a line of code must be interpreted consistently across all projects for which data are collected; (4) applications should be similar to work that is to be estimated—it makes little sense to use a baseline for batch information systems work to estimate a real-time, embedded application.

4.6.3 Metrics Collection, Computation, and Evaluation

KEY POINT

Baseline metrics data should be collected from a large, representative sampling of past software projects.

The process for establishing a baseline is illustrated in Figure 4.7. Ideally, data needed to establish a baseline has been collected in an ongoing manner. Sadly, this is rarely the case. Therefore, data collection requires a historical investigation of past projects to reconstruct required data. Once measures have been collected (unquestionably the most difficult step), metrics computation is possible. Depending on the breadth of measures collected, metrics can span a broad range of LOC or FP metrics as well as other quality- and project-oriented metrics. Finally, metrics must be evaluated and applied during estimation, technical work, project control, and process improvement. Metrics evaluation focuses on the underlying reasons for the results obtained and produces a set of indicators that guide the project or process.

4.7 MANAGING VARIATION: STATISTICAL PROCESS CONTROL

Because the software process and the product it produces both are influenced by many parameters (e.g., the skill level of practitioners, the structure of the software team, the knowledge of the customer, the technology that is to be implemented, the tools to be used in the development activity), metrics collected for one project or product will not be the same as similar metrics collected for another project. In fact, there is often significant variability in the metrics we collect as part of the software process.

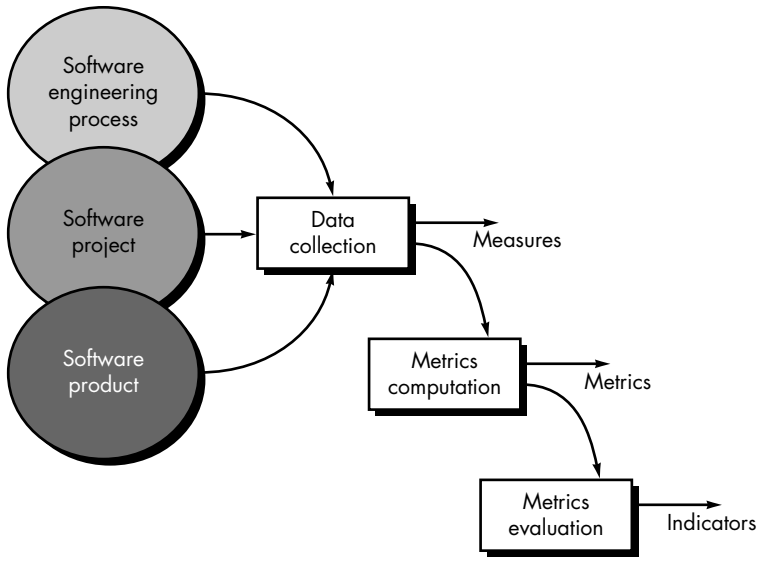


FIGURE 4.7 Software metrics collection process

Quote:

"If I had to reduce my message for management to just a few words, I'd say it all had to do with reducing variation."

W. Edwards Deming

Since the same process metrics will vary from project to project, how can we tell if improved (or degraded) metrics values that occur as consequence of improvement activities are having a quantitative impact? How do we know whether we're looking at a statistically valid trend or whether the "trend" is simply a result of statistical noise? When are changes (either positive or negative) to a particular software metric meaningful?

A graphical technique is available for determining whether changes and variation in metrics data are meaningful. Called the *control chart* and developed by Walter Shewart in the 1920s,[10] this technique enables individuals interested in software process improvement to determine whether the dispersion (variability) and "location" (moving average) of process metrics are *stable* (i.e., the process exhibits only natural or controlled changes) or *unstable* (i.e., the process exhibits out-of-control changes and metrics cannot be used to predict performance). Two different types of control charts are used in the assessment of metrics data [ZUL99]: (1) the moving range control chart and (2) the individual control chart.

? How can we be sure that the metrics we collect are statistically valid?

To illustrate the control chart approach, consider a software organization that collects the process metric, errors uncovered per review hour, E_r. Over the past 15 months, the organization has collected E_r for 20 small projects in the same general software development domain. The resultant values for E_r are represented in Figure 4.8. In the figure, E_r varies from a low of 1.2 for project 3 to a high of 5.9 for project 17. In an effort to improve the effectiveness of reviews, the software organization provided training and mentoring to all project team members beginning with project 11.

10 It should be noted that, although the control chart was originally developed for manufacturing processes, it is equally applicable for software processes.

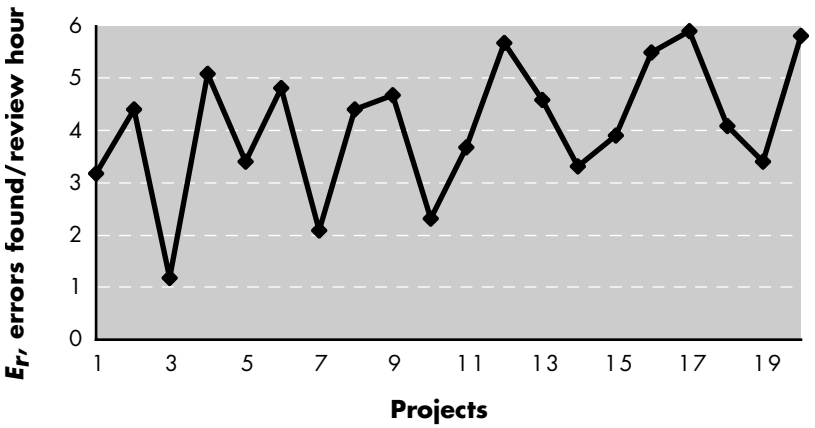


FIGURE 4.8 Metrics data for errors uncovered per review hour

Richard Zultner provides an overview of the procedure required to develop a *moving range* (mR) *control chart* for determining the stability of the process [ZUL99]:

1. Calculate the moving ranges: the absolute value of the successive differences between each pair of data points . . . Plot these moving ranges on your chart.
2. Calculate the mean of the moving ranges . . . plot this ("mR bar") as the center line on your chart.
3. Multiply the mean by 3.268. Plot this line as the *upper control limit* [UCL]. This line is three standard deviations above the mean.

Quote:

"If we can't tell signals from noise, how will we ever know if changes to the process are improvement—or illusions?"

Richard Zultner

Using the data represented in Figure 4.8 and the steps suggested by Zultner, we develop an mR control chart shown in Figure 4.9. The mR bar (mean) value for the moving range data is 1.71. The upper control limit is 5.58.

To determine whether the process metrics dispersion is stable, a simple question is asked: Are all the moving range values inside the UCL? For the example noted, the answer is "yes." Hence, the metrics dispersion is stable.

The individual control chart is developed in the following manner:[11]

1. Plot individual metrics values as shown in Figure 4.8.
2. Compute the average value, A_m, for the metrics values.
3. Multiply the mean of the mR values (the mR bar) by 2.660 and add A_m computed in step 2. This results in the *upper natural process limit* (UNPL). Plot the UNPL.
4. Multiply the mean of the mR values (the mR bar) by 2.660 and subtract this amount from A_m computed in step 2. This results in the *lower natural process limit* (LNPL). Plot the LNPL. If the LNPL is less than 0.0, it need not be plotted unless the metric being evaluated takes on values that are less than 0.0.
5. Compute a standard deviation as (UNPL $- A_m$)/3. Plot lines one and two standard deviations above and below A_m. If any of the standard deviation

11 The discussion that follows is a summary of steps suggested by Zultner [ZUL99].

FIGURE 4.9
Moving range control chart

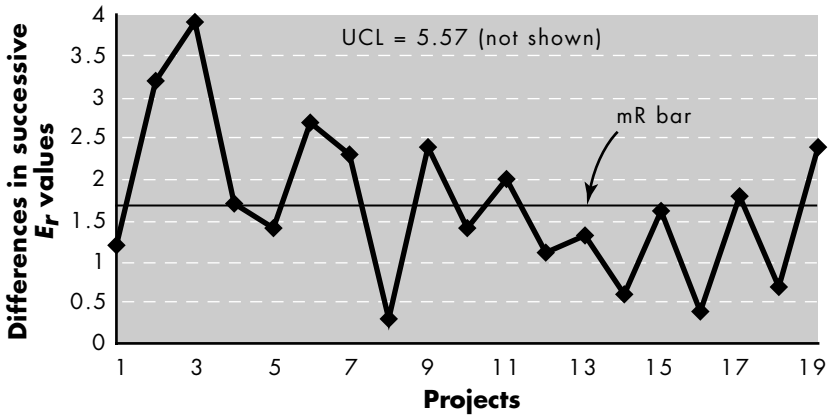
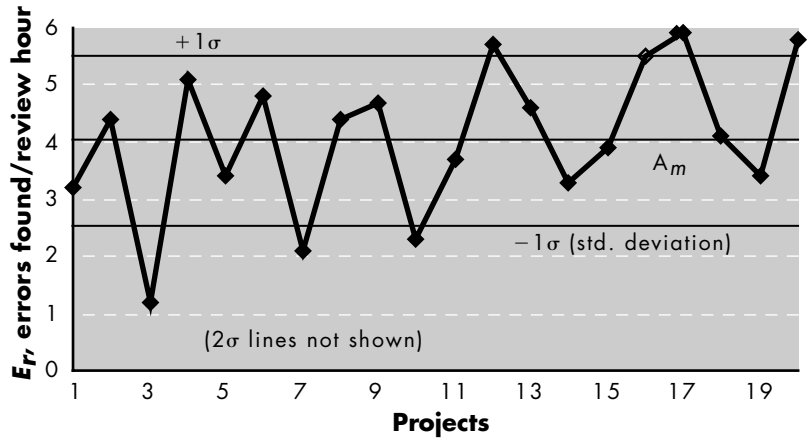


FIGURE 4.10
Individual control chart



lines is less than 0.0, it need not be plotted unless the metric being evaluated takes on values that are less than 0.0.

Applying these steps to the data represented in Figure 4.8, we derive an *individual control chart* as shown in Figure 4.10.

Zultner [ZUL99] reviews four criteria, called *zone rules,* that may be used to evaluate whether the changes represented by the metrics indicate a process that is in control or out of control. If any of the following conditions is true, the metrics data indicate a process that is out of control:

1. A single metrics value lies outside the UNPL.
2. Two out of three successive metrics values lie more than two standard deviations away from A_m.
3. Four out of five successive metrics values lie more than one standard deviation away from A_m.
4. Eight consecutive metrics values lie on one side of A_m.

WebRef

The *Common Control Chart Cookbook* covers the topic at some length and can be found at **www.sytsma.com/tqmtools/ctlchtprinciples.html**

Since all of these conditions fail for the values shown in Figure 4.10, the metrics data are derived from a stable process and trend information can be legitimately inferred from the metrics collected. Referring to Figure 4.10, it can be seen that the variability of E_r decreases after project 10 (i.e., after an effort to improve the effectiveness of reviews). By computing the mean value for the first 10 and last 10 projects, it can be shown that the mean value of E_r for projects 11–20 shows a 29 percent improvement over E_r for projects 1–10. Since the control chart indicates that the process is stable, it appears that efforts to improve review effectiveness are working.

4.8 METRICS FOR SMALL ORGANIZATIONS

If you're just starting to collect software metrics, remember to keep it simple. If you bury yourself with data, your metrics effort will fail.

The vast majority of software development organizations have fewer than 20 software people. It is unreasonable, and in most cases unrealistic, to expect that such organizations will develop comprehensive software metrics programs. However, it is reasonable to suggest that software organizations of all sizes measure and then use the resultant metrics to help improve their local software process and the quality and timeliness of the products they produce. Kautz [KAU99] describes a typical scenario that occurs when metrics programs are suggested for small software organizations:

> Originally, the software developers greeted our activities with a great deal of skepticism, but they eventually accepted them because we kept our measurements simple, tailored them to each organization, and ensured that they produced valuable information. In the end, the programs provided a foundation for taking care of customers and for planning and carrying out future work.

What Kautz suggests is a commonsense approach to the implementation of any software process related activity: keep it simple, customize to meet local needs, and be sure it adds value. In the paragraphs that follow, we examine how these guidelines relate to metrics for small shops.

How do I derive a set of "simple" software metrics?

"Keep it simple" is a guideline that works reasonably well in many activities. But how do we derive a "simple" set of software metrics that still provides value, and how can we be sure that these simple metrics will meet the needs of a particular software organization? We begin by focusing not on measurement but rather on results. The software group is polled to define a single objective that requires improvement. For example, "reduce the time to evaluate and implement change requests." A small organization might select the following set of easily collected measures:

- Time (hours or days) elapsed from the time a request is made until evaluation is complete, t_{queue}.
- Effort (person-hours) to perform the evaluation, W_{eval}.
- Time (hours or days) elapsed from completion of evaluation to assignment of change order to personnel, t_{eval}.

- Effort (person-hours) required to make the change, W_{change}.
- Time required (hours or days) to make the change, t_{change}.
- Errors uncovered during work to make change, E_{change}.
- Defects uncovered after change is released to the customer base, D_{change}.

Once these measures have been collected for a number of change requests, it is possible to compute the total elapsed time from change request to implementation of the change and the percentage of elapsed time absorbed by initial queuing, evaluation and change assignment, and change implementation. Similarly, the percentage of effort required for evaluation and implementation can be determined. These metrics can be assessed in the context of quality data, E_{change} and D_{change}. The percentages provide insight into where the change request process slows down and may lead to process improvement steps to reduce t_{queue}, W_{eval}, t_{eval}, W_{change}, and/or E_{change}. In addition, the defect removal efficiency can be computed as

$$\text{DRE} = E_{change} / (E_{change} + D_{change})$$

DRE can be compared to elapsed time and total effort to determine the impact of quality assurance activities on the time and effort required to make a change.

For small groups, the cost of collecting measures and computing metrics ranges from 3 to 8 percent of project budget during the learning phase and then drops to less than 1 percent of project budget after software engineers and project managers have become familiar with the metrics program [GRA99]. These costs can show a substantial return on investment if the insights derived from metrics data lead to meaningful process improvement for the software organization.

4.9 ESTABLISHING A SOFTWARE METRICS PROGRAM

The Software Engineering Institute has developed a comprehensive guidebook [PAR96] for establishing a "goal-driven" software metrics program. The guidebook suggests the following steps:

A Guidebook for Goal-Driven Software Measurement can be downloaded from **www.sei.cmu.edu**

1. Identify your business goals.
2. Identify what you want to know or learn.
3. Identify your subgoals.
4. Identify the entities and attributes related to your subgoals.
5. Formalize your measurement goals.
6. Identify quantifiable questions and the related indicators that you will use to help you achieve your measurement goals.
7. Identify the data elements that you will collect to construct the indicators that help answer your questions.
8. Define the measures to be used, and make these definitions operational.

9. Identify the actions that you will take to implement the measures.
10. Prepare a plan for implementing the measures.

A detailed discussion of these steps is best left to the SEI's guidebook. However, a brief overview of key points is worthwhile.

The software metrics you choose are driven by the business or technical goals you wish to accomplish.

Because software supports business functions, differentiates computer-based systems or products, or acts as a product in itself, goals defined for the business can almost always be traced downward to specific goals at the software engineering level. For example, consider a company that makes advanced home security systems which have substantial software content. Working as a team, software engineering and business managers can develop a list of prioritized business goals:

1. Improve our customers' satisfaction with our products.
2. Make our products easier to use.
3. Reduce the time it takes us to get a new product to market.
4. Make support for our products easier.
5. Improve our overall profitability.

The software organization examines each business goal and asks: "What activities do we manage or execute and what do we want to improve within these activities?" To answer these questions the SEI recommends the creation of an "entity-question list" in which all things (entities) within the software process that are managed or influenced by the software organization are noted. Examples of entities include development resources, work products, source code, test cases, change requests, software engineering tasks, and schedules. For each entity listed, software people develop a set of questions that assess quantitative characteristics of the entity (e.g., size, cost, time to develop). The questions derived as a consequence of the creation of an entity-question list lead to the derivation of a set of subgoals that relate directly to the entities created and the activities performed as part of the software process.

Consider the fourth goal: "Make support for our products easier." The following list of questions might be derived for this goal [PAR96]:

- Do customer change requests contain the information we require to adequately evaluate the change and then implement it in a timely manner?
- How large is the change request backlog?
- Is our response time for fixing bugs acceptable based on customer need?
- Is our change control process (Chapter 9) followed?
- Are high-priority changes implemented in a timely manner?

Based on these questions, the software organization can derive the following subgoal: Improve the performance of the change management process. The software

process entities and attributes that are relevant to the subgoal are identified and measurement goals associated with them are delineated.

The SEI [PAR96] provides detailed guidance for steps 6 through 10 of its goal-driven measurement approach. In essence, a process of stepwise refinement is applied in which goals are refined into questions that are further refined into entities and attributes that are then refined into metrics.

4.10 SUMMARY

Measurement enables managers and practitioners to improve the software process; assist in the planning, tracking, and control of a software project; and assess the quality of the product (software) that is produced. Measures of specific attributes of the process, project, and product are used to compute software metrics. These metrics can be analyzed to provide indicators that guide management and technical actions.

Process metrics enable an organization to take a strategic view by providing insight into the effectiveness of a software process. Project metrics are tactical. They enable a project manager to adapt project work flow and technical approach in a real-time manner.

Both size- and function-oriented metrics are used throughout the industry. Size-oriented metrics use the line of code as a normalizing factor for other measures such as person-months or defects. The function point is derived from measures of the information domain and a subjective assessment of problem complexity.

Software quality metrics, like productivity metrics, focus on the process, the project, and the product. By developing and analyzing a metrics baseline for quality, an organization can correct those areas of the software process that are the cause of software defects.

Metrics are meaningful only if they have been examined for statistical validity. The control chart is a simple method for accomplishing this and at the same time examining the variation and location of metrics results.

Measurement results in cultural change. Data collection, metrics computation, and metrics analysis are the three steps that must be implemented to begin a metrics program. In general, a goal-driven approach helps an organization focus on the right metrics for its business. By creating a metrics baseline—a database containing process and product measurements—software engineers and their managers can gain better insight into the work that they do and the product that they produce.

REFERENCES

[ALA97] Alain, A., M. Maya, J.M. Desharnais, and S. St. Pierre, "Adapting Function Points to Real-Time Software," *American Programmer,* vol. 10, no. 11, November 1997, pp. 32–43.

[ALB79] Albrecht, A.J., "Measuring Application Development Productivity," *Proc. IBM Application Development Symposium,* Monterey, CA, October 1979, pp. 83–92.

[ALB83] Albrecht, A.J. and J.E. Gaffney, "Software Function, Source Lines of Code and Development Effort Prediction: A Software Science Validation," *IEEE Trans. Software Engineering,* November 1983, pp. 639–648.

[ART85] Arthur, L.J., *Measuring Programmer Productivity and Software Quality,* Wiley-Interscience, 1985.

[BOE81] Boehm, B., *Software Engineering Economics,* Prentice-Hall, 1981.

[GRA87] Grady, R.B. and D.L. Caswell, *Software Metrics: Establishing a Company-wide Program,* Prentice-Hall, 1987.

[GRA92] Grady, R.G., *Practical Software Metrics for Project Management and Process Improvement,* Prentice-Hall, 1992.

[GRA94] Grady, R., "Successfully Applying Software Metrics," *Computer,* vol. 27, no. 9, September 1994, pp. 18–25.

[GRA99] Grable, R., et al., "Metrics for Small Projects: Experiences at SED," *IEEE Software,* March 1999, pp. 21–29.

[GIL88] Gilb, T., *Principles of Software Project Management,* Addison-Wesley, 1988.

[HET93] Hetzel, W., *Making Software Measurement Work,* QED Publishing Group, 1993.

[HUM95] Humphrey, W., *A Discipline for Software Engineering,* Addison-Wesley, 1995.

[IEE93] *IEEE Software Engineering Standards,* Standard 610.12-1990, pp. 47–48.

[IFP94] *Function Point Counting Practices Manual,* Release 4.0, International Function Point Users Group, 1994.

[JON86] Jones, C., *Programming Productivity,* McGraw-Hill, 1986.

[JON91] Jones, C., *Applied Software Measurement,* McGraw-Hill, 1991.

[JON98] Jones, C., *Estimating Software Costs,* McGraw-Hill, 1998.

[KAU99] Kautz, K., "Making Sense of Measurement for Small Organizations," *IEEE Software,* March 1999, pp. 14–20.

[MCC78] McCall, J.A. and J.P. Cavano, "A Framework for the Measurement of Software Quality," ACM Software Quality Assurance Workshop, November 1978.

[PAR96] Park, R.E., W.B. Goethert, and W.A. Florac, *Goal Driven Software Measurement—A Guidebook,* CMU/SEI-96-BH-002, Software Engineering Institute, Carnegie Mellon University, August 1996.

[PAU94] Paulish, D. and A. Carleton, "Case Studies of Software Process Improvement Measurement," *Computer,* vol. 27, no. 9, September 1994, pp. 50–57.

[RAG95] Ragland, B., "Measure, Metric or Indicator: What's the Difference?" *Crosstalk,* vol. 8, no. 3, March 1995, p. 29–30.

[TAJ81] Tajima, D. and T. Matsubara, "The Computer Software Industry in Japan," *Computer,* May 1981, p. 96.

[WHI95] Whitmire, S.A., "An Introduction to 3D Function Points", *Software Development,* April 1995, pp. 43–53.

[ZUL99] Zultner, R.E., "What Do Our Metrics Mean?" *Cutter IT Journal,* vol. 12, no. 4, April 1999, pp. 11–19.

PROBLEMS AND POINTS TO PONDER

4.1. Suggest three measures, three metrics, and corresponding indicators that might be used to assess an automobile.

4.2. Suggest three measures, three metrics, and corresponding indicators that might be used to assess the service department of an automobile dealership.

4.3. Describe the difference between process and project metrics in your own words.

4.4. Why should some software metrics be kept "private"? Provide examples of three metrics that should be private. Provide examples of three metrics that should be public.

4.5. Obtain a copy of Humphrey (*Introduction to the Personal Software Process,* Addison-Wesley, 1997) and write a one- or two-page summary that outlines the PSP approach.

4.6. Grady suggests an etiquette for software metrics. Can you add three more rules to those noted in Section 4.2.1?

4.7. Attempt to complete the fishbone diagram shown in Figure 4.3. That is, following the approach used for "incorrect" specifications, provide analogous information for "missing, ambiguous, and changed" specifications.

4.8. What is an indirect measure and why are such measures common in software metrics work?

4.9. Team A found 342 errors during the software engineering process prior to release. Team B found 184 errors. What additional measures would have to be made for projects A and B to determine which of the teams eliminated errors more efficiently? What metrics would you propose to help in making the determination? What historical data might be useful?

4.10. Present an argument against lines of code as a measure for software productivity. Will your case hold up when dozens or hundreds of projects are considered?

4.11. Compute the function point value for a project with the following information domain characteristics:

Number of user inputs: 32
Number of user outputs: 60
Number of user inquiries: 24
Number of files: 8
Number of external interfaces: 2

Assume that all complexity adjustment values are average.

4.12. Compute the 3D function point value for an embedded system with the following characteristics:

Internal data structures: 6
External data structure: 3

Number of user inputs: 12
Number of user outputs: 60
Number of user inquiries: 9
Number of external interfaces: 3
Transformations: 36
Transitions: 24

Assume that the complexity of these counts is evenly divided between low, average, and high.

4.13. The software used to control a photocopier requires 32,000 of C and 4,200 lines of Smalltalk. Estimate the number of function points for the software inside the photocopier.

4.14. McCall and Cavano (Section 4.5.1) define a "framework" for software quality. Using information contained in this and other books, expand each of the three major "points of view" into a set of quality factors and metrics.

4.15. Develop your own metrics (do not use those presented in this chapter) for correctness, maintainability, integrity, and usability. Be sure that they can be translated into quantitative values.

4.16. Is it possible for spoilage to increase while at the same time defects/KLOC decrease? Explain.

4.17. Does the LOC measure make any sense when fourth generation techniques are used? Explain.

4.18. A software organization has DRE data for 15 projects over the past two years. The values collected are 0.81, 0.71, 0.87, 0.54, 0.63, 0.71, 0.90, 0.82, 0.61, 0.84, 0.73, 0.88, 0.74, 0.86, 0.83. Create mR and individual control charts to determine whether these data can be used to assess trends.

FURTHER READINGS AND INFORMATION SOURCES

Software process improvement (SPI) has received a significant amount of attention over the past decade. Since measurement and software metrics are key to successfully improving the software process, many books on SPI also discuss metrics. Worthwhile additions to the literature include:

Burr, A. and M. Owen, *Statistical Methods for Software Quality,* International Thomson Publishing, 1996.

El Emam, K. and N. Madhavji (eds.), *Elements of Software Process Assessment and Improvement,* IEEE Computer Society, 1999.

Florac, W.A. and A.D. Carleton, *Measuring the Software Process: Statistical Process Control for Software Process Improvement,* Addison-Wesley, 1999.

Garmus, D. and D. Herron, *Measuring the Software Process: A Practical Guide to Functional Measurements,* Prentice-Hall, 1996.

Humphrey, W., *Introduction to the Team Software Process,* Addison-Wesley Longman, 2000.

Kan, S.H., *Metrics and Models in Software Quality Engineering,* Addison-Wesley, 1995.

Humphrey [HUM95], Yeh (*Software Process Control,* McGraw-Hill, 1993), Hetzel [HET93], and Grady [GRA92] discuss how software metrics can be used to provide the indicators necessary to improve the software process. Putnam and Myers (*Executive Briefing: Controlling Software Development,* IEEE Computer Society, 1996) and Pulford and his colleagues (*A Quantitative Approach to Software Management,* Addison-Wesley, 1996) discuss process metrics and their use from a management point of view.

Weinberg (*Quality Software Management,* Volume 2: *First Order Measurement,* Dorset House, 1993) presents a useful model for observing software projects, ascertaining the meaning of the observation, and determining its significance for tactical and strategic decisions. Garmus and Herron (*Measuring the Software Process,* Prentice-Hall, 1996) discuss process metrics with an emphasis on function point analysis. The Software Productivity Consortium (*The Software Measurement Guidebook,* Thomson Computer Press, 1995) provides useful suggestions for instituting an effective metrics approach. Oman and Pfleeger (*Applying Software Metrics,* IEEE Computer Society Press, 1997) have edited an excellent anthology of important papers on software metrics. Park, et al. [PAR96] have developed a detailed guidebook that provides step-by-step suggestions for instituting a software metrics program for software process improvement.

The newsletter *IT Metrics* (edited by Howard Rubin and published by Cutter Information Services) presents useful commentary on the state of software metrics in the industry. The magazines *Cutter IT Journal* and *Software Development* have regular articles and entire features dedicated to software metrics.

A wide variety of information sources on software process and project metrics are available on the Internet. An up-to-date list of World Wide Web references that are relevant to the software process and project metrics can be found at the SEPA Web site:

http://www.mhhe.com/engcs/compsci/pressman/resources/process-metrics.mhtml

CHAPTER

5 SOFTWARE PROJECT PLANNING

KEY CONCEPTS

automated tools. 139
decomposition techniques. 124
empirical models 132
estimation. 123
feasibility 117
make/buy decision. 136
outsourcing. 138
problem-based estimation. 126
process-based estimation. 130
resources 120
software scope . 115

Software project management begins with a set of activities that are collectively called *project planning*. Before the project can begin, the manager and the software team must estimate the work to be done, the resources that will be required, and the time that will elapse from start to finish. Whenever estimates are made, we look into the future and accept some degree of uncertainty as a matter of course. To quote Frederick Brooks [BRO75]:

> . . . our techniques of estimating are poorly developed. More seriously, they reflect an unvoiced assumption that is quite untrue, i.e., that all will go well. . . . because we are uncertain of our estimates, software managers often lack the courteous stubbornness to make people wait for a good product.

Although estimating is as much art as it is science, this important activity need not be conducted in a haphazard manner. Useful techniques for time and effort estimation do exist. Process and project metrics can provide historical perspective and powerful input for the generation of quantitative estimates. Past experience (of all people involved) can aid immeasurably as estimates are developed and reviewed. Because estimation lays a foundation for all other project planning activities and project planning provides the road map for successful software engineering, we would be ill-advised to embark without it.

QUICK LOOK

What is it? Software project planning actually encompasses all of the activities we discuss in Chapters 5 through 9. However, in the context of this chapter, planning involves estimation—your attempt to determine how much money, how much effort, how many resources, and how much time it will take to build a specific software-based system or product.

Who does it? Software managers—using information solicited from customers and software engineers and software metrics data collected from past projects.

Why is it important? Would you build a house without knowing how much you were about to spend? Of course not, and since most computer-based systems and products cost considerably more to build than a large house, it would seem reasonable to develop an estimate before you start creating the software.

What are the steps? Estimation begins with a description of the scope of the product. Until the scope is "bounded" it's not possible to develop a meaningful estimate. The problem is then decomposed into a set of smaller problems and each of these is estimated using historical data and experience as guides. It is advisable to generate your estimates using at least two different methods (as a cross check). Problem complexity and risk are considered before a final estimate is made.

What is the work product? A simple table delineating the tasks to be performed, the functions to be ▶

QUICK LOOK

implemented, and the cost, effort, and time involved for each is generated. A list of required project resources is also produced.

How do I ensure that I've done it right? That's hard, because you won't really know until the project has been completed. However, if you have experience and follow a systematic approach, generate estimates using solid historical data, create estimation data points using at least two different methods, and factor in complexity and risk, you can feel confident that you've given it your best shot.

5.1 OBSERVATIONS ON ESTIMATING

A leading executive was once asked what single characteristic was most important when selecting a project manager. His response: "a person with the ability to know what will go wrong before it actually does . . ." We might add: "and the courage to estimate when the future is cloudy."

Quote:

"Good estimating approaches and solid historical data offer the best hope that reality will win over impossible demands."

Capers Jones

Estimation of resources, cost, and schedule for a software engineering effort requires experience, access to good historical information, and the courage to commit to quantitative predictions when qualitative information is all that exists. Estimation carries inherent risk[1] and this risk leads to uncertainty.

Project complexity has a strong effect on the uncertainty inherent in planning. Complexity, however, is a relative measure that is affected by familiarity with past effort. The first-time developer of a sophisticated e-commerce application might consider it to be exceedingly complex. However, a software team developing its tenth e-commerce Web site would consider such work run of the mill. A number of quantitative software complexity measures have been proposed [ZUS97]. Such measures are applied at the design or code level and are therefore difficult to use during software planning (before a design and code exist). However, other, more subjective assessments of complexity (e.g., the function point complexity adjustment factors described in Chapter 4) can be established early in the planning process.

Project size is another important factor that can affect the accuracy and efficacy of estimates. As size increases, the interdependency among various elements of the software grows rapidly.[2] Problem decomposition, an important approach to estimating, becomes more difficult because decomposed elements may still be formidable. To paraphrase Murphy's law: "What can go wrong will go wrong"—and if there are more things that can fail, more things will fail.

KEY POINT

Project complexity, project size, and the degree of structural uncertainty all affect the reliability of estimates.

The *degree of structural uncertainty* also has an effect on estimation risk. In this context, structure refers to the degree to which requirements have been solidified, the ease with which functions can be compartmentalized, and the hierarchical nature of the information that must be processed.

---

1 Systematic techniques for risk analysis are presented in Chapter 6.

2 Size often increases due to the "scope creep" that occurs when the customer changes requirements. Increases in project size can have a geometric impact on project cost and schedule (M. Mah, personal communication).

Quote:

"It is the mark of an instructed mind to rest satisfied with the degree of precision which the nature of a subject admits, and not to seek exactness when only an approximation of the truth is possible."

Aristotle

The availability of historical information has a strong influence on estimation risk. By looking back, we can emulate things that worked and improve areas where problems arose. When comprehensive software metrics (Chapter 4) are available for past projects, estimates can be made with greater assurance, schedules can be established to avoid past difficulties, and overall risk is reduced.

Risk is measured by the degree of uncertainty in the quantitative estimates established for resources, cost, and schedule. If project scope is poorly understood or project requirements are subject to change, uncertainty and risk become dangerously high. The software planner should demand completeness of function, performance, and interface definitions (contained in a *System Specification*). The planner, and more important, the customer should recognize that variability in software requirements means instability in cost and schedule.

However, a project manager should not become obsessive about estimation. Modern software engineering approaches (e.g., evolutionary process models) take an iterative view of development. In such approaches, it is possible[3] to revisit the estimate (as more information is known) and revise it when the customer makes changes to requirements.

5.2 PROJECT PLANNING OBJECTIVES

The more you know, the better you estimate. Therefore, update your estimates as the project progresses.

The objective of software project planning is to provide a framework that enables the manager to make reasonable estimates of resources, cost, and schedule. These estimates are made within a limited time frame at the beginning of a software project and should be updated regularly as the project progresses. In addition, estimates should attempt to define best case and worst case scenarios so that project outcomes can be bounded.

The planning objective is achieved through a process of information discovery that leads to reasonable estimates. In the following sections, each of the activities associated with software project planning is discussed.

5.3 SOFTWARE SCOPE

The first activity in software project planning is the determination of software scope. Function and performance allocated to software during system engineering (Chapter 10) should be assessed to establish a project scope that is unambiguous and understandable at the management and technical levels. A statement of software scope must be *bounded.*

Software scope describes the data and control to be processed, function, performance, constraints, interfaces, and reliability. Functions described in the statement

3 This is not meant to imply that it is always politically acceptable to modify initial estimates. A mature software organization and its managers recognize that change is not free. And yet, many customers demand (incorrectly) that an estimate once made must be maintained regardless of changing circumstances.

of scope 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.

5.3.1 Obtaining Information Necessary for Scope

Things are always somewhat hazy at the beginning of a software project. A need has been defined and basic goals and objectives have been enunciated, but the information necessary to define scope (a prerequisite for estimation) has not yet been delineated.

The most commonly used technique to bridge the communication gap between the customer and developer and to get the communication process started is to conduct a preliminary meeting or interview. The first meeting between the software engineer (the analyst) and the customer can be likened to the awkwardness of a first date between two adolescents. Neither person knows what to say or ask; both are worried that what they do say will be misinterpreted; both are thinking about where it might lead (both likely have radically different expectations here); both want to get the thing over with; but at the same time, both want it to be a success.

? How should we initiate communication between the developer and the customer?

Yet, communication must be initiated. Gause and Weinberg [GAU89] suggest that the analyst start by asking *context-free questions;* that is, a set of questions that will lead to a basic understanding of the problem, the people who want a solution, the nature of the solution desired, and the effectiveness of the first encounter itself.

The first set of context-free questions focuses on the customer, the overall goals and benefits. For example, the analyst might ask:

- Who is behind the request for this work?
- Who will use the solution?
- What will be the economic benefit of a successful solution?
- Is there another source for the solution?

The next set of questions enables the analyst to gain a better understanding of the problem and the customer to voice any perceptions about a solution:

- How would you (the customer) characterize "good" output that would be generated by a successful solution?
- What problem(s) will this solution address?
- Can you show me (or describe) the environment in which the solution will be used?
- Will any special performance issues or constraints affect the way the solution is approached?

The final set of questions focuses on the effectiveness of the meeting. Gause and Weinberg call these "meta-questions" and propose the following (abbreviated) list:

- Are you the right person to answer these questions? Are answers "official"?
- Are my questions relevant to the problem that you have?
- Am I asking too many questions?
- Can anyone else provide additional information?
- Should I be asking you anything else?

These questions (and others) will help to "break the ice" and initiate the communication that is essential to establish the scope of the project. But a question and answer meeting format is not an approach that has been overwhelmingly successful. In fact, the Q&A session should be used for the first encounter only and then be replaced by a meeting format that combines elements of problem solving, negotiation, and specification.

Customers and software engineers often have an unconscious "us and them" mindset. Rather than working as a team to identify and refine requirements, each constituency defines its own "territory" and communicates through a series of memos, formal position papers, documents, and question and answer sessions. History has shown that this approach works poorly. Misunderstandings abound, important information is omitted, and a successful working relationship is never established.

XRef

Requirements elicitation techniques are discussed in Chapter 11.

With these problems in mind, a number of independent investigators have developed a team-oriented approach to requirements gathering that can be applied to help establish the scope of a project. Called *facilitated application specification techniques* (FAST), this approach encourages the creation of a joint team of customers and developers who work together to identify the problem, propose elements of the solution, negotiate different approaches, and specify a preliminary set of requirements.

5.3.2 Feasibility

Quote:

"It's 106 miles to Chicago, we got a full tank of gas, half a pack of cigarettes, it's dark and we're wearing sunglasses. Hit it."

The Blues Brothers

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 customers), only to become mired in a project that is doomed from the onset. Putnam and Myers [PUT97a] address this issue when they write:

> . . . not 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?

For some projects in established areas the answers are easy. You have done projects like this one before. After a few hours or sometimes a few weeks of investigation, you are sure you can do it again.

Projects on the margins of your experience are not so easy. A team may have to spend several months discovering what the central, difficult-to-implement requirements of a new application actually are. Do some of these requirements pose risks that would make the project infeasible? Can these risks be overcome? The feasibility team ought to carry initial architecture and design of the high-risk requirements to the point at which it can answer these questions. In some cases, when the team gets negative answers, a reduction in requirements may be negotiated.

Technical feasibility is important, but business need is even more important. It does no good to build a high tech system or product that no one really wants.

Meantime, the cartoon people [senior managers] are drumming their fingers nervously on their large desks. Often, they wave their fat cigars in a lordly manner and yell impatiently through the smoke screen, "Enough. Do it!"

Many of the projects that appear in the newspapers a few years later as whopping failures got started this way.

Putnam and Myers correctly suggest that scoping is not enough. Once scope is understood, the software team and others 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.

5.3.3 A Scoping Example

Communication with the customer leads to a definition of the data and control that are processed, the functions that must be implemented, the performance and constraints that bound the system, and related information. As an example, consider software for a conveyor line sorting system (CLSS). The statement of scope for CLSS follows:

The conveyor line sorting system (CLSS) sorts boxes moving along a conveyor line. Each box is identified by a bar code that contains a part number and is sorted into one of six bins at the end of the line. The boxes pass by a sorting station that contains a bar code reader and a PC. The sorting station PC is connected to a shunting mechanism that sorts the boxes into the bins. Boxes pass in random order and are evenly spaced. The line is moving at five feet per minute. CLSS is depicted schematically in Figure 5.1.

CLSS software receives input information from a bar code reader at time intervals that conform to the conveyor line speed. Bar code data will be decoded into box identification format. The software will do a look-up in a part number database containing a maximum of 1000 entries to determine proper bin location for the box currently at the reader (sorting station). The proper bin location is passed to a sorting shunt that will position boxes in the appropriate bin. A record of the bin destination for each box will be maintained for later recovery and reporting. CLSS software will also receive input from a pulse tachometer that will be used to synchronize the control signal to the shunting mechanism. Based on the number of pulses generated between the sorting station and the shunt, the software will produce a control signal to the shunt to properly position the box.

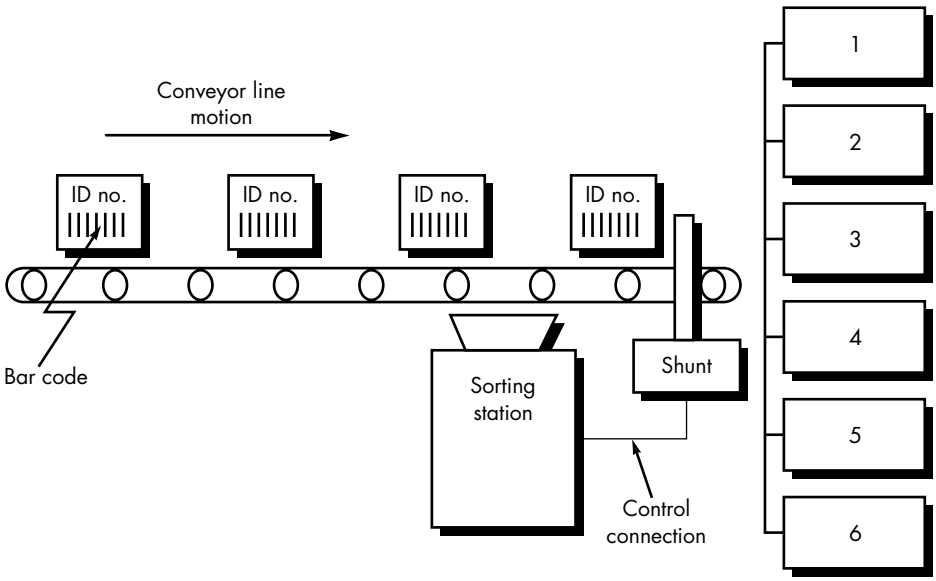


FIGURE 5.1
A conveyor line sorting system

The project planner examines the statement of scope and extracts all important software functions. This process, called *decomposition,* was discussed in Chapter 3 and results in the following functions:[4]

- Read bar code input.
- Read pulse tachometer.
- Decode part code data.
- Do database look-up.
- Determine bin location.
- Produce control signal for shunt.
- Maintain record of box destinations.

Adjust estimates to reflect difficult performance requirements and design constraints, even if scope is otherwise simple.

In this case, performance is dictated by conveyor line speed. Processing for each box must be completed before the next box arrives at the bar code reader. The CLSS software is constrained by the hardware it must access (the bar code reader, the shunt, the PC), the available memory, and the overall conveyor line configuration (evenly spaced boxes).

Function, performance, and constraints must be evaluated together. The same function can precipitate an order of magnitude difference in development effort when considered in the context of different performance bounds. The effort and cost required

4 In reality, the functional decomposition is performed during system engineering (Chapter 10). The planner uses information derived from the *System Specification* to define software functions.

to develop CLSS software would be dramatically different if function remains the same (i.e., put boxes into bins) but performance varies. For instance, if the conveyor line average speed increases by a factor of 10 (performance) and boxes are no long spaced evenly (a constraint), software would become considerably more complex—thereby requiring more effort. Function, performance, and constraints are intimately connected.

KEY POINT

A consideration of software scope must include an evaluation of all external interfaces.

Software interacts with other elements of a computer-based system. The planner considers the nature and complexity of each interface to determine any effect on development resources, cost, and schedule. The concept of an interface is interpreted to include (1) the hardware (e.g., processor, peripherals) that executes the software and devices (e.g., machines, displays) indirectly controlled by the software, (2) software that already exists (e.g., database access routines, reusable software components, operating system) and must be linked to the new software, (3) people that make use of the software via keyboard or other I/O devices, and (4) procedures that precede or succeed the software as a sequential series of operations. In each case, the information transfer across the interface must be clearly understood.

The least precise aspect of software scope is a discussion of reliability. Software reliability measures do exist (see Chapter 8) but they are rarely used at this stage of a project. Classic hardware reliability characteristics like *mean-time-between-failures* (MTBF) can be difficult to translate to the software domain. However, the general nature of the software may dictate special considerations to ensure "reliability." For example, software for an air traffic control system or the space shuttle (both human-rated systems) must not fail or human life may be lost. An inventory control system or word-processor software should not fail, but the impact of failure is considerably less dramatic. Although it may not be possible to quantify software reliability as precisely as we would like in the statement of scope, we can use the nature of the project to aid in formulating estimates of effort and cost to assure reliability.

? What is the primary source of information for determining scope?

If a *System Specification* (see Chapter 10) has been properly developed, nearly all information required for a description of software scope is available and documented before software project planning begins. In cases where a specification has not been developed, the planner must take on the role of system analyst to determine attributes and bounds that will influence estimation tasks.

5.4 RESOURCES

The second software planning task is estimation of the resources required to accomplish the software development effort. Figure 5.2 illustrates development resources as a pyramid. The *development environment*—hardware and software tools—sits at the foundation of the resources pyramid and provides the infrastructure to support the development effort. At a higher level, we encounter reusable *software components*—software building blocks that can dramatically reduce development costs and accelerate delivery. At the top of the pyramid is the primary resource—*people*. Each resource is specified with four characteristics: description of the resource, a state-

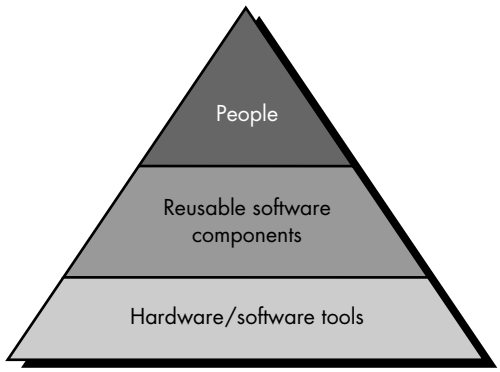


FIGURE 5.2 Project resources

ment of availability, time when the resource will be required; duration of time that 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.

5.4.1 Human Resources

XRef

The roles software people play and the team organizations that they populate are discussed in Chapter 3.

The planner begins by evaluating 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 specified. For relatively small projects (one person-year or less), a single individual may perform all software engineering tasks, consulting with specialists as required.

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.

5.4.2 Reusable Software Resources

KEY POINT

To be reused effectively, software components must be cataloged, standardized, and validated.

Component-based software engineering (CBSE)[5] emphasizes reusability—that is, the creation and reuse of software building blocks [HOO91]. Such building blocks, often called *components,* must be cataloged for easy reference, standardized for easy application, and validated for easy integration.

Bennatan [BEN92] 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 that has been developed internally for 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

5 Component-based software engineering is considered in detail in Chapter 27.

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 that must be built by the software team specifically for the needs of the current project.

The following guidelines should be considered by the software planner when reusable components are specified as a resource:

What issues should we consider when we plan to reuse existing software components?

1. If off-the-shelf components meet project requirements, acquire them. The cost for acquisition and integration of off-the-shelf components will almost always be less than the cost to develop equivalent software.[6] In addition, risk is relatively low.
2. If full-experience components are available, the risks associated with modification and integration are generally acceptable. The project plan should reflect the use of these components.
3. If partial-experience components are available, their use for the current project must be analyzed. If extensive modification is required before the components can be properly integrated with other elements of the software, proceed carefully—risk is high. The cost to modify partial-experience components can sometimes be greater than the cost to develop new components.

Ironically, reusable software components are often neglected during planning, only to become a paramount concern during the development phase of 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.

5.4.3 Environmental Resources

The environment that supports the 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.[7] Because most software

6 When existing software components are used during a project, the overall cost reduction can be dramatic. In fact, industry data indicate that cost, time to market, and the number of defects delivered to the field all are reduced.

7 Other hardware—the target environment—is the computer on which the software will execute when it has been released to the end-user.

organizations have multiple constituencies that require access to the SEE, a project planner 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 numerical control (NC) used on a class of machine tools may require a specific machine tool (e.g., an NC lathe) as part of the validation test step; a software project for advanced page-layout may need a digital-typesetting system at some point during development. Each hardware element must be specified by the software project planner.

5.5 SOFTWARE PROJECT ESTIMATION

Quote:

"In an age of outsourcing and increased competition, the ability to estimate more accurately . . . has emerged as a critical survival factor for many IT groups."

Rob Thomsett

In the early days of computing, software costs constituted a small percentage of the overall computer-based system cost. An order of magnitude error in estimates of software cost had relatively little impact. Today, software is the most expensive element of virtually all computer-based systems. For complex, custom systems, a large cost estimation error can make the difference between profit and loss. Cost overrun can be disastrous for the developer.

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% 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, we should recognize that the longer we wait, the more we know, and the more we know, the less likely we are to make serious errors in our 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 SEE, deadlines) are 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" 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

$$d = f(v_i)$$

where d is one of a number of estimated values (e.g., effort, cost, project duration) and v_i are selected independent parameters (e.g., estimated LOC or FP).

Automated estimation tools implement one or more decomposition techniques or empirical models. When combined with a graphical user interface, automated tools 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.

Estimation tools

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 4, we examined the characteristics of some of the software metrics that provide the basis for historical estimation data.

5.6 DECOMPOSITION TECHNIQUES

Software project estimation is a form of problem solving, and in most cases, the problem to be solved (i.e., developing a cost and effort estimate for a software project) is too complex to be considered in one piece. For this reason, we decompose the problem, recharacterizing it as a set of smaller (and hopefully, more manageable) problems.

In Chapter 3, the decomposition approach was discussed from two different points of view: decomposition of the problem and decomposition of the process. Estimation uses one or both forms of partitioning. But before an estimate can be made, the project planner must understand the scope of the software to be built and generate an estimate of its "size."

5.6.1 Software Sizing

The "size" of software to be built can be estimated using a direct measure, LOC, or an indirect measure, FP.

The accuracy of a software project estimate is predicated on a number of things: (1) the degree to which the planner has properly estimated the size of the product to be built; (2) the ability to translate the size estimate into human effort, calendar time, and dollars (a function of the availability of reliable software metrics from past projects); (3) the degree to which the project plan reflects the abilities of the software team; and (4) the stability of product requirements and the environment that supports the software engineering effort.

In this section, we consider the *software sizing* problem. Because a project estimate is only as good as the estimate of the size of the work to be accomplished, sizing represents the project planner's first major challenge. In the context of project planning, size refers to a quantifiable outcome of the software project. If a direct approach is taken, size can be measured in LOC. If an indirect approach is chosen, size is represented as FP.

Putnam and Myers [PUT92] suggest four different approaches to the sizing problem:

? How do we size the software that we're planning to build?

"Fuzzy logic" sizing. This approach uses the approximate reasoning techniques that are the cornerstone of fuzzy logic. To apply this approach, the planner must identify the type of application, establish its magnitude on a qualitative scale, and then refine the magnitude within the original range. Although personal experience can be used, the planner should also have access to a historical database of projects[8] so that estimates can be compared to actual experience.

Function point sizing. The planner develops estimates of the information domain characteristics discussed in Chapter 4.

Standard component sizing. Software is composed of a number of different "standard components" that are generic to a particular application area. For example, the standard components for an information system are subsystems, modules, screens, reports, interactive programs, batch programs, files, LOC, and object-level instructions. The project planner estimates the number of occurrences of each standard component and then uses historical project data to determine the delivered size per standard component. To illustrate, consider an information systems application. The planner estimates that 18 reports will be generated. Historical data indicates that 967 lines of COBOL [PUT92] are required per report. This enables the planner to estimate that 17,000 LOC will be required for the reports component. Similar estimates and computation are made for other standard components, and a combined size value (adjusted statistically) results.

Change sizing. This approach is used when a project encompasses the use of existing software that must be modified in some way as part of a project. The planner estimates the number and type (e.g., reuse, adding code, changing code, deleting code) of modifications that must be accomplished. Using an "effort ratio" [PUT92] for each type of change, the size of the change may be estimated.

Putnam and Myers suggest that the results of each of these sizing approaches be combined statistically to create a *three-point* or *expected value* estimate. This is accomplished by developing optimistic (low), most likely, and pessimistic (high) values for size and combining them using Equations (5-1) described in the next section.

---

8 See Section 5.9 for a discussion of estimating tools that make use of a historical database and the other sizing techniques discussed in this section..

5.6.2 Problem-Based Estimation

In Chapter 4, lines of code and function points were described as measures from which productivity metrics can be computed. LOC and FP data are used in two ways during software project estimation: (1) as an estimation variable to "size" each element of the software and (2) as baseline metrics collected from past projects and used in conjunction with estimation variables to develop cost and effort projections.

? What do LOC- and FP-oriented estimation have in common?

LOC and FP estimation are distinct estimation techniques. Yet both have a number of characteristics in common. The project planner begins with a bounded statement of software scope and from this statement attempts to decompose software into problem functions that can each be estimated individually. LOC or FP (the estimation variable) is then estimated for each function. Alternatively, the planner may choose another component for sizing such as classes or objects, changes, or business processes affected.

Baseline productivity metrics (e.g., LOC/pm or FP/pm[9]) are then applied to the appropriate estimation variable, and cost or effort for the function is derived. Function estimates are combined to produce an overall estimate for the entire project.

ADVICE

When collecting productivity metrics for projects, be sure to establish a taxonomy of project types. This will enable you to compute domain-specific averages, making estimation more accurate.

It is important to note, however, that there is often substantial scatter in productivity metrics for an organization, making the use of a single baseline productivity metric suspect. In general, LOC/pm or FP/pm averages should be computed by project domain. That is, projects should be grouped by team size, application area, complexity, and other relevant parameters. Local domain averages should then be computed. When a new project is estimated, it should first be allocated to a domain, and then the appropriate domain average for productivity should be used in generating the estimate.

The LOC and FP estimation techniques differ in the level of detail required for decomposition and the target of the partitioning. When LOC is used as the estimation variable, decomposition[10] is absolutely essential and is often taken to considerable levels of detail. The following decomposition approach has been adapted from Phillips [PHI98]:[11]

KEY POINT

For LOC estimates, decomposition focuses on software functions.

```
define product scope;
identify functions by decomposing scope;
do while functions remain
        select a function_j
        assign all functions to subfunctions list;
```

9 The acronym *pm* stands for person-month.

10 In general, problem functions are decomposed. However, a list of standard components (Section 5.6.1) may be used instead.

11 The informal process design language noted here is intended to illustrate the general approach for sizing. It does not consider every logical contingency.

```
        do while subfunctions remain
        select subfunction_k
        if subfunction_k resembles subfunction_d described in a historical data base
        then    note historical cost, effort, size (LOC or FP) data for subfunction_d;
                adjust historical cost, effort, size data based on any differences;
                use adjusted cost, effort, size data to derive partial estimate, E_p;
                project estimate = sum of {E_p};
        else    if cost, effort, size (LOC or FP) for subfunction_k can be estimated
                then derive partial estimate, E_p;
                project estimate = sum of {E_p};
                else subdivide subfunction_k into smaller subfunctions;
                add these to subfunctions list;
                endif
        endif
        enddo
enddo
```

This decomposition approach assumes that all functions can be decomposed into subfunctions that will resemble entries in a historical data base. If this is not the case, then another sizing approach must be applied. The greater the degree of partitioning, the more likely reasonably accurate estimates of LOC can be developed.

For FP estimates, decomposition focuses on information domain characteristics.

For FP estimates, decomposition works differently. Rather than focusing on function, each of the information domain characteristics—inputs, outputs, data files, inquiries, and external interfaces—as well as the 14 complexity adjustment values discussed in Chapter 4 are estimated. The resultant estimates can then be used to derive a FP value that can be tied to past data and used to generate an estimate.

Regardless of the estimation variable that is used, the project planner begins by estimating a range of values for each function or information domain value. Using historical data or (when all else fails) intuition, the planner estimates an optimistic, most likely, and pessimistic size value for each function or count for each information domain value. An implicit indication of the degree of uncertainty is provided when a range of values is specified.

A three-point or expected value can then be computed. The *expected value* for the estimation variable (size), *S,* can be computed as a weighted average of the optimistic (s_{opt}), most likely (s_m), and pessimistic (s_{pess}) estimates. For example,

How do I compute the expected value for software size?

$$S = (s_{opt} + 4s_m + s_{pess})/6 \quad (5\text{-}1)$$

gives heaviest credence to the "most likely" estimate and follows a beta probability distribution. We assume that there is a very small probability the actual size result will fall outside the optimistic or pessimistic values.

Once the expected value for the estimation variable has been determined, historical LOC or FP productivity data are applied. Are the estimates correct? The only reasonable answer to this question is: "We can't be sure." Any estimation technique, no matter how sophisticated, must be cross-checked with another approach. Even then, common sense and experience must prevail.

5.6.3 An Example of LOC-Based Estimation

As an example of LOC and FP problem-based estimation techniques, let us consider a software package to be developed for a computer-aided design application for mechanical components. A review of the *System Specification* indicates that the software is to execute on an engineering workstation and must interface with various computer graphics peripherals including a mouse, digitizer, high resolution color display and laser printer.

Using the *System Specification* as a guide, a preliminary statement of software scope can be developed:

> The CAD software will accept two- and three-dimensional geometric data from an engineer. The engineer will interact and control the CAD system through a user interface that will exhibit characteristics of good human/machine interface design. All geometric data and other supporting information will be maintained in a CAD database. Design analysis modules will be developed to produce the required output, which will be displayed on a variety of graphics devices. The software will be designed to control and interact with peripheral devices that include a mouse, digitizer, laser printer, and plotter.

This statement of scope is preliminary—it is *not* bounded. Every sentence would have to be expanded to provide concrete detail and quantitative bounding. For example, before estimation can begin the planner must determine what "characteristics of good human/machine interface design" means or what the size and sophistication of the "CAD database" are to be.

Many modern applications reside on a network or are part of a client/server architecture. Therefore, be sure that your estimates include the effort required for the development of "infrastructure" software.

For our purposes, we assume that further refinement has occurred and that the following major software functions are identified:

- User interface and control facilities (UICF)
- Two-dimensional geometric analysis (2DGA)
- Three-dimensional geometric analysis (3DGA)
- Database management (DBM)
- Computer graphics display facilities (CGDF)
- Peripheral control function (PCF)
- Design analysis modules (DAM)

Following the decomposition technique for LOC, an estimation table, shown in Figure 5.3, is developed. A range of LOC estimates is developed for each function. For example, the range of LOC estimates for the 3D geometric analysis function is optimistic—4600 LOC, most likely—6900 LOC, and pessimistic—8600 LOC.

FIGURE 5.3 Estimation table for the LOC method

Function	Estimated LOC
User interface and control facilities (UICF)	2,300
Two-dimensional geometric analysis (2DGA)	5,300
Three-dimensional geometric analysis (3DGA)	6,800
Database management (DBM)	3,350
Computer graphics display facilities (CGDF)	4,950
Peripheral control function (PCF)	2,100
Design analysis modules (DAM)	8,400
Estimated lines of code	*33,200*

Do not succumb to the temptation to use this result as your estimate. You should derive another estimate using a different method.

Applying Equation (5-1), the expected value for the 3D geometric analysis function is 6800 LOC. Other estimates are derived in a similar fashion. By summing vertically in the estimated LOC column, an estimate of 33,200 lines of code is established for the CAD system.

A review of historical data indicates that the organizational average productivity for systems of this type is 620 LOC/pm. Based on a burdened labor rate of $8000 per month, the cost per line of code is approximately $13. Based on the LOC estimate and the historical productivity data, the total estimated project cost is $431,000 and the estimated effort is 54 person-months.[12]

5.6.4 An Example of FP-Based Estimation

Decomposition for FP-based estimation focuses on information domain values rather than software functions. Referring to the function point calculation table presented in Figure 5.4, the project planner estimates inputs, outputs, inquiries, files, and external interfaces for the CAD software. For the purposes of this estimate, the complexity weighting factor is assumed to be average. Figure 5.4 presents the results of this estimate.

Information on FP cost estimating tools can be obtained at **www.spr.com**

FIGURE 5.4 Estimating information domain values

Information domain value	Opt.	Likely	Pess.	Est. count	Weight	FP count
Number of inputs	20	24	30	24	4	97
Number of outputs	12	15	22	16	5	78
Number of inquiries	16	22	28	22	5	88
Number of files	4	4	5	4	10	42
Number of external interfaces	2	2	3	2	7	15
Count total						*320*

12 Estimates are rounded-off to the nearest $1,000 and person-month. Arithmetic precision to the nearest dollar or tenth of a month is unrealistic.

Each of the complexity weighting factors is estimated and the complexity adjustment factor is computed as described in Chapter 4:

Factor	Value
Backup and recovery	4
Data communications	2
Distributed processing	0
Performance critical	4
Existing operating environment	3
On-line data entry	4
Input transaction over multiple screens	5
Master files updated on-line	3
Information domain values complex	5
Internal processing complex	5
Code designed for reuse	4
Conversion/installation in design	3
Multiple installations	5
Application designed for change	5
Complexity adjustment factor	1.17

Finally, the estimated number of FP is derived:

$$FP_{\text{estimated}} = \text{count-total} \times [0.65 + 0.01 \times \Sigma (F_i)]$$
$$FP_{\text{estimated}} = 375$$

The organizational average productivity for systems of this type is 6.5 FP/pm. Based on a burdened labor rate of $8000 per month, the cost per FP is approximately $1230. Based on the LOC estimate and the historical productivity data, the total estimated project cost is $461,000 and the estimated effort is 58 person-months.

5.6.4 Process-Based Estimation

The most common technique for estimating a project is to base the estimate on the process that will be used. That is, the process is decomposed into a relatively small set of tasks and the effort required to accomplish each task is estimated.

XRef

A common process framework (CPF) is discussed in Chapter 2.

Like the problem-based techniques, process-based estimation begins with a delineation of software functions obtained from the project scope. A series of software process activities must be performed for each function. Functions and related software process activities may be represented as part of a table similar to the one presented in Figure 3.2.

Once problem functions and process activities are melded, the planner estimates the effort (e.g., person-months) that will be required to accomplish each software process activity for each software function. These data constitute the central matrix of the table in Figure 3.2. Average labor rates (i.e., cost/unit effort) are then applied to the effort estimated for each process activity. It is very likely the labor rate will vary for each task. Senior staff heavily involved in early activities are generally more expensive than junior staff involved in later design tasks, code generation, and early testing.

FIGURE 5.5
Process-based estimation table

Activity →	CC	Planning	Risk analysis	Engineering		Construction release		CE	Totals
Task →				Analysis	Design	Code	Test		
Function									
↓									
UICF				0.50	2.50	0.40	5.00	n/a	8.40
2DGA				0.75	4.00	0.60	2.00	n/a	7.35
3DGA				0.50	4.00	1.00	3.00	n/a	8.50
CGDF				0.50	3.00	1.00	1.50	n/a	6.00
DBM				0.50	3.00	0.75	1.50	n/a	5.75
PCF				0.25	2.00	0.50	1.50	n/a	4.25
DAM				0.50	2.00	0.50	2.00	n/a	5.00
Totals	0.25	0.25	0.25	3.50	20.50	4.50	16.50		46.00
% effort	1%	1%	1%	8%	45%	10%	36%		

CC = customer communication CE = customer evaluation

If time permits, use greater granularity when specifying tasks in Figure 5.5, such as breaking analysis into its major tasks and estimating each separately.

Costs and effort for each function and software process activity are computed as the last step. If process-based estimation is performed independently of LOC or FP estimation, we now have two or three estimates for cost and effort that may be compared and reconciled. If both sets of estimates show reasonable agreement, there is good reason to believe that the estimates are reliable. If, on the other hand, the results of these decomposition techniques show little agreement, further investigation and analysis must be conducted.

5.6.5 An Example of Process-Based Estimation

To illustrate the use of process-based estimation, we again consider the CAD software introduced in Section 5.6.3. The system configuration and all software functions remain unchanged and are indicated by project scope.

Referring to the completed process-based table shown in Figure 5.5, estimates of effort (in person-months) for each software engineering activity are provided for each CAD software function (abbreviated for brevity). The engineering and construction release activities are subdivided into the major software engineering tasks shown. Gross estimates of effort are provided for customer communication, planning, and risk analysis. These are noted in the total row at the bottom of the table. Horizontal and vertical totals provide an indication of estimated effort required for analysis, design, code, and test. It should be noted that 53 percent of all effort is expended on front-end engineering tasks (requirements analysis and design), indicating the relative importance of this work.

Based on an average burdened labor rate of $8,000 per month, the total estimated project cost is $368,000 and the estimated effort is 46 person-months. If desired, labor rates could be associated with each software process activity or software engineering task and computed separately.

Do not expect that all estimates will agree within a percent or two. If the estimates are within a 20 percent band, they can be reconciled into a single value.

Total estimated effort for the CAD software range from a low of 46 person-months (derived using a process-based estimation approach) to a high of 58 person-months (derived using an FP estimation approach). The average estimate (using all three approaches) is 53 person-months. The maximum variation from the average estimate is approximately 13 percent.

What happens when agreement between estimates is poor? The answer to this question requires a re-evaluation of information used to make the estimates. Widely divergent estimates can often be traced to one of two causes:

1. The scope of the project is not adequately understood or has been misinterpreted by the planner.
2. Productivity data used for problem-based estimation techniques is inappropriate for the application, obsolete (in that it no longer accurately reflects the software engineering organization), or has been misapplied.

The planner must determine the cause of divergence and then reconcile the estimates.

5.7 EMPIRICAL ESTIMATION MODELS

An estimation model reflects the population of projects from which it has been derived. Therefore, the model is domain sensitive.

An *estimation model* for computer software uses empirically derived formulas to predict effort as a function of LOC or FP. Values for LOC or FP are estimated using the approach described in Sections 5.6.2 and 5.6.3. But instead of using the tables described in those sections, the resultant values for LOC or FP are plugged into the estimation model.

The empirical data that support most estimation models are derived from a limited sample of projects. For this reason, no estimation model is appropriate for all classes of software and in all development environments. Therefore, the results obtained from such models must be used judiciously.[13]

5.7.1 The Structure of Estimation Models

A typical estimation model is derived using regression analysis on data collected from past software projects. The overall structure of such models takes the form [MAT94]

$$E = A + B \times (ev)^C \tag{5-2}$$

where *A, B,* and *C* are empirically derived constants, *E* is effort in person-months, and *ev* is the estimation variable (either LOC or FP). In addition to the relationship noted in Equation (5-2), the majority of estimation models have some form of project adjust-

13 In general, an estimation model should be calibrated for local conditions. The model should be run using the results of completed projects. Data predicted by the model should be compared to actual results and the efficacy of the model (for local conditions) should be assessed. If agreement is not good, model coefficients and exponents must be recomputed using local data.

ment component that enables *E* to be adjusted by other project characteristics (e.g., problem complexity, staff experience, development environment). Among the many LOC-oriented estimation models proposed in the literature are

$E = 5.2 \times (\text{KLOC})^{0.91}$	Walston-Felix model
$E = 5.5 + 0.73 \times (\text{KLOC})^{1.16}$	Bailey-Basili model
$E = 3.2 \times (\text{KLOC})^{1.05}$	Boehm simple model
$E = 5.288 \times (\text{KLOC})^{1.047}$	Doty model for KLOC > 9

None of these models should be used without careful calibration to your environment.

FP-oriented models have also been proposed. These include

$E = -13.39 + 0.0545\ \text{FP}$	Albrecht and Gaffney model
$E = 60.62 \times 7.728 \times 10^{-8}\ \text{FP}^3$	Kemerer model
$E = 585.7 + 15.12\ \text{FP}$	Matson, Barnett, and Mellichamp model

A quick examination of these models indicates that each will yield a different result[14] for the same values of LOC or FP. The implication is clear. Estimation models must be calibrated for local needs!

5.7.2 The COCOMO Model

In his classic book on "software engineering economics," Barry Boehm [BOE81] introduced a hierarchy of software estimation models bearing the name COCOMO, for *COnstructive COst MOdel.* The original COCOMO model became one of the most widely used and discussed software cost estimation models in the industry. It has evolved into a more comprehensive estimation model, called *COCOMO II* [BOE96, BOE00]. Like its predecessor, COCOMO II is actually a hierarchy of estimation models that address the following areas:

Detailed information on COCOMO II, including downloadable software, can be obtained at **sunset.usc.edu/COCOMOII/cocomo.html**

Application composition model. Used during the early stages of software engineering, when prototyping of user interfaces, consideration of software and system interaction, assessment of performance, and evaluation of technology maturity are paramount.

Early design stage model. Used once requirements have been stabilized and basic software architecture has been established.

Post-architecture-stage model. Used during the construction of the software.

Like all estimation models for software, the COCOMO II models require sizing information. Three different sizing options are available as part of the model hierarchy: object points, function points, and lines of source code.

The COCOMO II application composition model uses object points and is illustrated in the following paragraphs. It should be noted that other, more

14 Part of the reason is that these models are often derived from relatively small populations of projects in only a few application domains.

TABLE 5.1 Complexity weighting for object types [BOE96]

Object type	Complexity weight		
	Simple	Medium	Difficult
Screen	1	2	3
Report	2	5	8
3GL component			10

sophisticated estimation models (using FP and KLOC) are also available as part of COCOMO II.

What is an "object point"?

Like function points (Chapter 4), the *object point* is an indirect software measure that is computed using counts of the number of (1) screens (at the user interface), (2) reports, and (3) components likely to be required to build the application. Each object instance (e.g., a screen or report) is classified into one of three complexity levels (i.e., simple, medium, or difficult) using criteria suggested by Boehm [BOE96]. In essence, complexity is a function of the number and source of the client and server data tables that are required to generate the screen or report and the number of views or sections presented as part of the screen or report.

Once complexity is determined, the number of screens, reports, and components are weighted according to Table 5.1. The object point count is then determined by multiplying the original number of object instances by the weighting factor in Table 5.1 and summing to obtain a total object point count. When component-based development or general software reuse is to be applied, the percent of reuse (%reuse) is estimated and the object point count is adjusted:

$$\text{NOP} = (\text{object points}) \times [(100 - \%\text{reuse})/100]$$

where NOP is defined as new object points.

To derive an estimate of effort based on the computed NOP value, a "productivity rate" must be derived. Table 5.2 presents the productivity rate

$$\text{PROD} = \text{NOP}/\text{person-month}$$

TABLE 5.2 Productivity rates for object points [BOE96]

Developer's experience/capability	Very low	Low	Nominal	High	Very high
Environment maturity/capability	Very low	Low	Nominal	High	Very high
PROD	4	7	13	25	50

for different levels of developer experience and development environment maturity. Once the productivity rate has been determined, an estimate of project effort can be derived as

estimated effort = NOP/PROD

In more advanced COCOMO II models,[15] a variety of scale factors, cost drivers, and adjustment procedures are required. A complete discussion of these is beyond the scope of this book. The interested reader should see [BOE00] or visit the COCOMO II Web site.

5.7.3 The Software Equation

The software equation [PUT92] is a dynamic multivariable model that assumes a specific distribution of effort over the life of a software development project. The model has been derived from productivity data collected for over 4000 contemporary software projects. Based on these data, an estimation model of the form

$$E = [\text{LOC} \times B^{0.333}/P]^3 \times (1/t^4) \tag{5-3}$$

where E = effort in person-months or person-years
t = project duration in months or years
B = "special skills factor"[16]
P = "productivity parameter" that reflects:

- Overall process maturity and management practices
- The extent to which good software engineering practices are used
- The level of programming languages used
- The state of the software environment
- The skills and experience of the software team
- The complexity of the application

Information on software cost estimation tools that have evolved from the software equation can be obtained at **www.qsm.com**

Typical values might be P = 2,000 for development of real-time embedded software; P = 10,000 for telecommunication and systems software; P = 28,000 for business systems applications.[17] The productivity parameter can be derived for local conditions using historical data collected from past development efforts.

It is important to note that the software equation has two independent parameters: (1) an estimate of size (in LOC) and (2) an indication of project duration in calendar months or years.

---

15 As noted earlier, these models use FP and KLOC counts for the size variable.

16 B increases slowly as "the need for integration, testing, quality assurance, documentation, and management skills grow [PUT92]." For small programs (KLOC = 5 to 15), B = 0.16. For programs greater than 70 KLOC, B = 0.39.

17 It is important to note that the productivity parameter can be empirically derived from local project data.

To simplify the estimation process and use a more common form for their estimation model, Putnam and Myers [PUT92] suggest a set of equations derived from the software equation. Minimum development time is defined as

$$t_{min} = 8.14\ (LOC/P)^{0.43} \text{ in months for } t_{min} > 6 \text{ months} \quad (5\text{-}4a)$$

$$E = 180\ Bt^3 \text{ in person-months for } E \geq 20 \text{ person-months} \quad (5\text{-}4b)$$

Note that t in Equation (5-4b) is represented in years.

Using Equations (5-4) with $P = 12{,}000$ (the recommended value for scientific software) for the CAD software discussed earlier in this chapter,

$$t_{min} = 8.14\ (33200/12000)^{0.43}$$
$$t_{min} = 12.6 \text{ calendar months}$$
$$E = 180 \times 0.28 \times (1.05)^3$$
$$E = 58 \text{ person-months}$$

The results of the software equation correspond favorably with the estimates developed in Section 5.6. Like the COCOMO model noted in the preceding section, the software equation has evolved over the past decade. Further discussion of an extended version of this estimation approach can be found in [PUT97b].

5.8 THE MAKE/BUY DECISION

In many software application areas, it is often more cost effective to acquire than develop computer software. Software engineering managers are faced with a *make/buy decision* that can be further complicated by a number of acquisition options: (1) software may be purchased (or licensed) off-the-shelf, (2) "full-experience" or "partial-experience" software components (see Section 5.4.2) may be acquired and then modified and integrated to meet specific needs, or (3) software may be custom built by an outside contractor to meet the purchaser's specifications.

There are times when off-the-shelf software provides a "perfect" solution except for a few special features that you can't live without. In many cases, it's worth living without the special features!

The steps involved in the acquisition of software are defined by the criticality of the software to be purchased and the end cost. In some cases (e.g., low-cost PC software), it is less expensive to purchase and experiment than to conduct a lengthy evaluation of potential software packages. For more expensive software products, the following guidelines can be applied:

1. Develop specifications for function and performance of the desired software. Define measurable characteristics whenever possible.

2. Estimate the internal cost to develop and the delivery date.

3a. Select three or four candidate applications that best meet your specifications.

3b. Select reusable software components that will assist in constructing the required application.

4. Develop a comparison matrix that presents a head-to-head comparison of key functions. Alternatively, conduct benchmark tests to compare candidate software.
5. Evaluate each software package or component based on past product quality, vendor support, product direction, reputation, and the like.
6. Contact other users of the software and ask for opinions.

In the final analysis, the make/buy decision is made based on the following conditions: (1) Will the delivery date of the software product be sooner than that for internally developed software? (2) Will the cost of acquisition plus the cost of customization be less than the cost of developing the software internally? (3) Will the cost of outside support (e.g., a maintenance contract) be less than the cost of internal support? These conditions apply for each of the acquisition options.

? Is there a systematic way to sort through the options associated with the make/buy decision?

5.8.1 Creating a Decision Tree

The steps just described can be augmented using statistical techniques such as *decision tree analysis* [BOE89]. For example, Figure 5.6 depicts a decision tree for a software-based system, *X*. In this case, the software engineering organization can (1) build system *X* from scratch, (2) reuse existing "partial-experience" components to construct the system, (3) buy an available software product and modify it to meet local needs, or (4) contract the software development to an outside vendor.

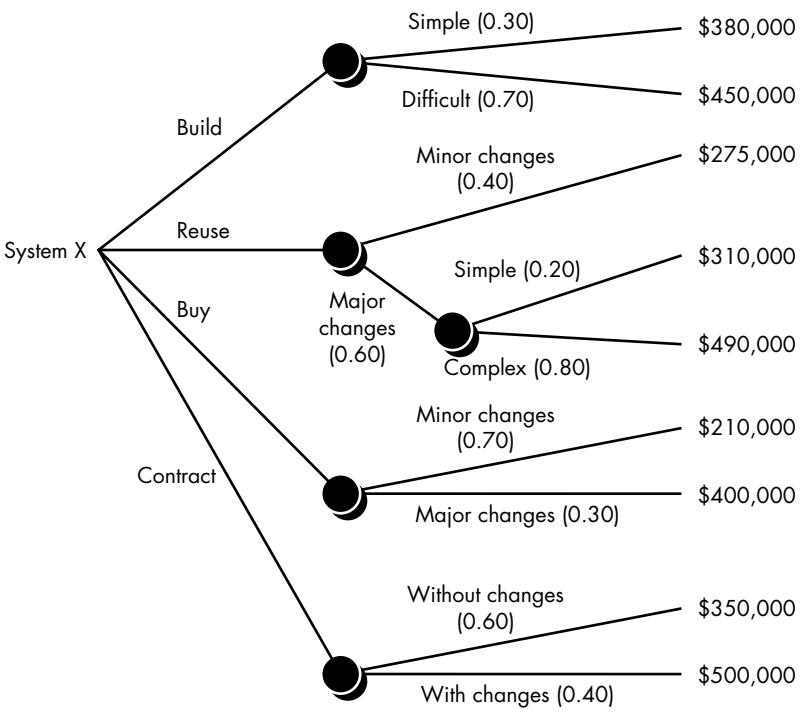


FIGURE 5.6 A decision tree to support the make/buy decision

If the system is to be built from scratch, there is a 70 percent probability that the job will be difficult. Using the estimation techniques discussed earlier in this chapter, the project planner projects that a difficult development effort will cost $450,000. A "simple" development effort is estimated to cost $380,000. The expected value for cost, computed along any branch of the decision tree, is

$$\text{expected cost} = \Sigma \text{ (path probability)}_i \times \text{(estimated path cost)}_i$$

where i is the decision tree path. For the build path,

$$\text{expected cost}_{\text{build}} = 0.30 (\$380K) + 0.70 (\$450K) = \$429K$$

Following other paths of the decision tree, the projected costs for reuse, purchase and contract, under a variety of circumstances, are also shown. The expected costs for these paths are

$$\text{expected cost}_{\text{reuse}} = 0.40 (\$275K) + 0.60 [0.20(\$310K) + 0.80(\$490K)] = \$382K$$
$$\text{expected cost}_{\text{buy}} = 0.70(\$210K) + 0.30(\$400K)] = \$267K$$
$$\text{expected cost}_{\text{contract}} = 0.60(\$350K) + 0.40(\$500K)] = \$410K$$

Based on the probability and projected costs that have been noted in Figure 5.6, the lowest expected cost is the "buy" option.

It is important to note, however, that many criteria—not just cost— must be considered during the decision-making process. Availability, experience of the developer/vendor/contractor, conformance to requirements, local "politics," and the likelihood of change are but a few of the criteria that may affect the ultimate decision to build, reuse, buy, or contract.

WebRef

An excellent tutorial on decision tree analysis can be found at **www.demon.co.uk/mindtool/dectree.html**

5.8.2 Outsourcing

Quote:

"As a rule, outsourcing requires even more skillful management than in-house development."

Steve McConnell

Sooner or later, every company that develops computer software asks a fundamental question: "Is there a way that we can get the software and systems we need at a lower price?" The answer to this question is not a simple one, and the emotional discussions that occur in response to the question always lead to a single word: outsourcing.

In concept, outsourcing is extremely simple. Software engineering activities are contracted to a third party who does the work at lower cost and, hopefully, higher quality. Software work conducted within a company is reduced to a contract management activity.

The decision to outsource can be either strategic or tactical. At the strategic level, business managers consider whether a significant portion of all software work can be contracted to others. At the tactical level, a project manager determines whether part or all of a project can be best accomplished by subcontracting the software work.

Regardless of the breadth of focus, the outsourcing decision is often a financial one. A detailed discussion of the financial analysis for outsourcing is beyond the

scope of this book and is best left to others (e.g., [MIN95]). However, a brief review of the pros and cons of the decision is worthwhile.

Useful information (papers, pointers) on outsourcing can be found at **www.outsourcing.com**

On the positive side, cost savings can usually be achieved by reducing the number of software people and the facilities (e.g., computers, infrastructure) that support them. On the negative side, a company loses some control over the software that it needs. Since software is a technology that differentiates its systems, services, and products, a company runs the risk of putting the fate of its competitiveness into the hands of a third party.

The trend toward outsourcing will undoubtedly continue. The only way to blunt the trend is to recognize that software work is extremely competitive at all levels. The only way to survive is to become as competitive as the outsourcing vendors themselves.

5.9 AUTOMATED ESTIMATION TOOLS

The decomposition techniques and empirical estimation models described in the preceding sections are available as part of a wide variety of software tools. These automated estimation tools allow the planner to estimate cost and effort and to perform "what-if" analyses for important project variables such as delivery date or staffing. Although many automated estimation tools exist, all exhibit the same general characteristics and all perform the following six generic functions [JON96]:

Estimation tools

1. **Sizing of project deliverables.** The "size" of one or more software work products is estimated. Work products include the external representation of software (e.g., screen, reports), the software itself (e.g., KLOC), functionality delivered (e.g., function points), descriptive information (e.g. documents).
2. **Selecting project activities.** The appropriate process framework (Chapter 2) is selected and the software engineering task set is specified.
3. **Predicting staffing levels.** The number of people who will be available to do the work is specified. Because the relationship between people available and work (predicted effort) is highly nonlinear, this is an important input.
4. **Predicting software effort.** Estimation tools use one or more models (e.g., Section 5.7) that relate the size of the project deliverables to the effort required to produce them.
5. **Predicting software cost.** Given the results of step 4, costs can be computed by allocating labor rates to the project activities noted in step 2.
6. **Predicting software schedules.** When effort, staffing level, and project activities are known, a draft schedule can be produced by allocating labor across software engineering activities based on recommended models for effort distribution (Chapter 7).

When different estimation tools are applied to the same project data, a relatively large variation in estimated results is encountered. More important, predicted values sometimes are significantly different than actual values. This reinforces the notion that the output of estimation tools should be used as one "data point" from which estimates are derived—not as the only source for an estimate.

5.10 SUMMARY

The software project planner must estimate three things before a project begins: how long it will take, how much effort will be required, and how many people will be involved. In addition, the planner must predict the resources (hardware and software) that will be required and the risk involved.

The statement of scope helps the planner to develop estimates using one or more techniques that fall into two broad categories: decomposition and empirical modeling. Decomposition techniques require a delineation of major software functions, followed by estimates of either (1) the number of LOC, (2) selected values within the information domain, (3) the number of person-months required to implement each function, or (4) the number of person-months required for each software engineering activity. Empirical techniques use empirically derived expressions for effort and time to predict these project quantities. Automated tools can be used to implement a specific empirical model.

Accurate project estimates generally use at least two of the three techniques just noted. By comparing and reconciling estimates derived using different techniques, the planner is more likely to derive an accurate estimate. Software project estimation can never be an exact science, but a combination of good historical data and systematic techniques can improve estimation accuracy.

REFERENCES

[BEN92] Bennatan, E.M., *Software Project Management: A Practitioner's Approach,* McGraw-Hill, 1992.

[BOE81] Boehm, B., *Software Engineering Economics,* Prentice-Hall, 1981.

[BOE89] Boehm, B., *Risk Management,* IEEE Computer Society Press, 1989.

[BOE96] Boehm, B., "Anchoring the Software Process," *IEEE Software,* vol. 13, no. 4, July 1996, pp. 73–82.

[BOE00] Boehm, B., et al., *Software Cost Estimation in COCOMO II,* Prentice-Hall, 2000.

[BRO75] Brooks, F., *The Mythical Man-Month,* Addison-Wesley, 1975.

[GAU89] Gause, D.C. and G.M. Weinberg, *Exploring Requirements: Quality Before Design,* Dorset House, 1989.

[HOO91] Hooper, J. and R.O. Chester, *Software Reuse: Guidelines and Methods,* Plenum Press, 1991.

[JON96] Jones, C., "How Software Estimation Tools Work," *American Programmer,* vol. 9, no. 7, July 1996, pp. 19–27.
[MAT94] Matson, J., B. Barrett, and J. Mellichamp, "Software Development Cost Estimation Using Function Points," *IEEE Trans. Software Engineering,* vol. SE-20, no. 4, April 1994, pp. 275–287.
[MIN95] Minoli, D., *Analyzing Outsourcing,* McGraw-Hill, 1995.
[PHI98] Phillips, D., *The Software Project Manager's Handbook,* IEEE Computer Society Press, 1998.
[PUT92] Putnam, L. and W. Myers, *Measures for Excellence,* Yourdon Press, 1992.
[PUT97a] Putnam, L. and W. Myers, "How Solved Is the Cost Estimation Problem?" *IEEE Software,* November 1997, pp. 105–107.
[PUT97b] Putnam, L. and W. Myers, *Industrial Strength Software: Effective Management Using Measurement,* IEEE Computer Society Press, 1997.
[ZUS97] Zuse, H., *A Framework for Software Measurement,* deGruyter, 1997.

PROBLEMS AND POINTS TO PONDER

5.1. Assume that you are the project manager for a company that builds software for consumer products. You have been contracted to build the software for a home security system. Write a statement of scope that describes the software. Be sure your statement of scope is bounded. If you're unfamiliar with home security systems, do a bit of research before you begin writing. *Alternate:* Replace the home security system with another problem that is of interest to you.

5.2. Software project complexity is discussed briefly in Section 5.1. Develop a list of software characteristics (e.g., concurrent operation, graphical output) that affect the complexity of a project. Prioritize the list.

5.3. Performance is an important consideration during planning. Discuss how performance can be interpreted differently depending upon the software application area.

5.4. Do a functional decomposition of the home security system software you described in problem 5.1. Estimate the size of each function in LOC. Assuming that your organization produces 450 LOC/pm with a burdened labor rate of $7000 per person-month, estimate the effort and cost required to build the software using the LOC-based estimation technique described in Section 5.6.3.

5.5. Using the 3D function point measure described in Chapter 4, compute the number of FP for the home security system software and derive effort and cost estimates using the FP-based estimation technique described in Section 5.6.4.

5.6. Use the COCOMO II model to estimate the effort required to build software for a simple ATM that produces 12 screens, 10 reports, and will require approximately

80 software components. Assume average complexity and average developer/environment maturity. Use the application composition model with object points.

5.7. Use the software equation to estimate the home security system software. Assume that Equations (5-4) are applicable and that $P = 8000$.

5.8. Compare the effort estimates derived in problems 5.4, 5.5, and 5.7. Develop a single estimate for the project using a three-point estimate. What is the standard deviation and how does it affect your degree of certainty about the estimate?

5.9. Using the results obtained in problem 5.8, determine whether it's reasonable to expect that the software can be built within the next six months and how many people would have to be used to get the job done.

5.10. Develop a spreadsheet model that implements one or more of the estimation techniques described in this chapter. Alternatively, acquire one or more on-line models for estimation from Web-based sources.

5.11. *For a project team,* develop a software tool that implements each of the estimation techniques developed in this chapter.

5.12. It seems odd that cost and schedule estimates are developed during software project planning—before detailed software requirements analysis or design has been conducted. Why do you think this is done? Are there circumstances when it should not be done?

5.13. Recompute the expected values noted for the decision tree in Figure 5.6 assuming that every branch has a 50–50 probability. Would this change your final decision?

FURTHER READINGS AND INFORMATION SOURCES

Most software project management books contain discussions of project estimation. Jones (*Estimating Software Costs,* McGraw-Hill, 1998) has written the most comprehensive treatment of the subject published to date. His book contains models and data that are applicable to software estimating in every application domain. Roetzheim and Beasley (*Software Project Cost and Schedule Estimating: Best Practices,* Prentice-Hall, 1997) present many useful models and suggest step-by-step guidelines for generating the best possible estimates.

Phillips [PHI98], Bennatan (*On Time, Within Budget: Software Project Management Practices and Techniques,* Wiley, 1995), Whitten (*Managing Software Development Projects: Formula for Success,* Wiley, 1995), Wellman (*Software Costing,* Prentice-Hall, 1992), and Londeix (*Cost Estimation for Software Development,* Addison-Wesley, 1987) contain useful information on software project planning and estimation.

Putnam and Myer's detailed treatment of software cost estimating ([PUT92] and [PUT97b]) and Boehm's books on software engineering economics ([BOE81] and

COCOMO II [BOE00]) describe empirical estimation models. These books provide detailed analysis of data derived from hundreds of software projects. An excellent book by DeMarco (*Controlling Software Projects,* Yourdon Press, 1982) provides valuable insight into the management, measurement, and estimation of software projects. Sneed (*Software Engineering Management,* Wiley, 1989) and Macro (*Software Engineering: Concepts and Management,* Prentice-Hall, 1990) consider software project estimation in considerable detail.

Lines-of-code cost estimation is the most commonly used approach in the industry. However, the impact of the object-oriented paradigm (see Part Four) may invalidate some estimation models. Lorenz and Kidd (*Object-Oriented Software Metrics,* Prentice-Hall, 1994) and Cockburn (*Surviving Object-Oriented Projects,* Addison-Wesley, 1998) consider estimation for object-oriented systems.

A wide variety of information sources on software planning and estimation is available on the Internet. An up-to-date list of World Wide Web references that are relevant to software estimation can be found at the SEPA Web site:
http://www.mhhe.com/engcs/compsci/pressman/resources/project-plan.mhtml

CHAPTER

6 RISK ANALYSIS AND MANAGEMENT

KEY CONCEPTS

assessment 154
components and drivers 149
identification . . . 148
mitigation 156
monitoring 157
projection 151
refinement 156
risk exposure . . 153
risk strategies . . 146
risk table 151
RMMM plan 159
safety and hazards 158

In his book on risk analysis and management, Robert Charette [CHA89] presents a conceptual definition of risk:

> First, risk concerns future happenings. Today and yesterday are beyond active concern, as we are already reaping what was previously sowed by our past actions. The question is, can we, therefore, by changing our actions today, create an opportunity for a different and hopefully better situation for ourselves tomorrow. This means, second, that risk involves change, such as in changes of mind, opinion, actions, or places . . . [Third,] risk involves choice, and the uncertainty that choice itself entails. Thus paradoxically, risk, like death and taxes, is one of the few certainties of life.

When risk is considered in the context of software engineering, Charette's three conceptual underpinnings are always in evidence. The future is our concern—what risks might cause the software project to go awry? Change is our concern—how will changes in customer requirements, development technologies, target computers, and all other entities connected to the project affect timeliness and overall success? Last, we must grapple with choices—what methods and tools should we use, how many people should be involved, how much emphasis on quality is "enough"?

QUICK LOOK

What is it? Risk analysis and management are a series of steps that help a software team to understand and manage uncertainty. Many problems can plague a software project. A risk is a potential problem—it might happen, it might not. But, regardless of the outcome, it's a really good idea to identify it, assess its probability of occurrence, estimate its impact, and establish a contingency plan should the problem actually occur.

Who does it? Everyone involved in the software process—managers, software engineers, and customers—participate in risk analysis and management.

Why is it important? Think about the Boy Scout motto: "Be prepared." Software is a difficult undertaking. Lots of things can go wrong, and frankly, many often do. It's for this reason that being prepared—understanding the risks and taking proactive measures to avoid or manage them—is a key element of good software project management.

What are the steps? Recognizing what can go wrong is the first step, called "risk identification." Next, each risk is analyzed to determine the likelihood that it will occur and the damage that it will do if it does occur. Once this information is established, risks are ranked, by probability and impact. Finally, a plan is developed to manage those risks with high probability and high impact.

What is the work product? A risk mitigation, monitoring, and management (RMMM) plan or ▶

QUICK LOOK

How do I ensure that I've done it right? The risks that are analyzed and managed should be derived from thorough study of the people, the product, the process, and the project. The RMMM should be revisited as the project proceeds to ensure that risks are kept up to date. Contingency plans for risk management should be realistic.

Peter Drucker [DRU75] once said, "While it is futile to try to eliminate risk, and questionable to try to minimize it, it is essential that the risks taken be the right risks." Before we can identify the "right risks" to be taken during a software project, it is important to identify all risks that are obvious to both managers and practitioners.

6.1 REACTIVE VS. PROACTIVE RISK STRATEGIES

Reactive risk strategies have been laughingly called the "Indiana Jones school of risk management" [THO92]. In the movies that carried his name, Indiana Jones, when faced with overwhelming difficulty, would invariably say, "Don't worry, I'll think of something!" Never worrying about problems until they happened, Indy would react in some heroic way.

Quote:

"If you don't actively attack the risks, they will actively attack you."

Tom Gilb

Sadly, the average software project manager is not Indiana Jones and the members of the software project team are not his trusty sidekicks. Yet, the majority of software teams rely solely on reactive risk strategies. At best, a reactive strategy monitors the project for likely risks. Resources are set aside to deal with them, should they become actual problems. More commonly, the software team does nothing about risks until something goes wrong. Then, the team flies into action in an attempt to correct the problem rapidly. This is often called a *fire fighting mode.* When this fails, "crisis management" [CHA92] takes over and the project is in real jeopardy.

A considerably more intelligent strategy for risk management is to be proactive. A proactive strategy begins long before technical work is initiated. Potential risks are identified, their probability and impact are assessed, and they are ranked by importance. Then, the software team establishes a plan for managing risk. The primary objective is to avoid risk, but because not all risks can be avoided, the team works to develop a contingency plan that will enable it to respond in a controlled and effective manner. Throughout the remainder of this chapter, we discuss a proactive strategy for risk management.

6.2 SOFTWARE RISKS

Although there has been considerable debate about the proper definition for software risk, there is general agreement that risk always involves two characteristics [HIG95]:

- *Uncertainty*—the risk may or may not happen; that is, there are no 100% probable risks.[1]
- *Loss*—if the risk becomes a reality, unwanted consequences or losses will occur.

When risks are analyzed, it is important to quantify the level of uncertainty and the degree of loss associated with each risk. To accomplish this, different categories of risks are considered.

? What types of risks are we likely to encounter as the software is built?

Project risks threaten the project plan. That is, if project risks become real, it is likely that project schedule will slip and that costs will increase. Project risks identify potential budgetary, schedule, personnel (staffing and organization), resource, customer, and requirements problems and their impact on a software project. In Chapter 5, project complexity, size, and the degree of structural uncertainty were also defined as project (and estimation) risk factors.

Technical risks threaten the quality and timeliness of the software to be produced. If a technical risk becomes a reality, implementation may become difficult or impossible. Technical risks identify potential design, implementation, interface, verification, and maintenance problems. In addition, specification ambiguity, technical uncertainty, technical obsolescence, and "leading-edge" technology are also risk factors. Technical risks occur because the problem is harder to solve than we thought it would be.

Business risks threaten the viability of the software to be built. Business risks often jeopardize the project or the product. Candidates for the top five business risks are (1) building a excellent product or system that no one really wants (market risk), (2) building a product that no longer fits into the overall business strategy for the company (strategic risk), (3) building a product that the sales force doesn't understand how to sell, (4) losing the support of senior management due to a change in focus or a change in people (management risk), and (5) losing budgetary or personnel commitment (budget risks). It is extremely important to note that simple categorization won't always work. Some risks are simply unpredictable in advance.

Quote:

"[Today,] no one has the luxury of getting to know a task so well that it holds no surprises, and surprises mean risk."

Stephen Grey

Another general categorization of risks has been proposed by Charette [CHA89]. *Known risks* are those that can be uncovered after careful evaluation of the project plan, the business and technical environment in which the project is being developed, and other reliable information sources (e.g., unrealistic delivery date, lack of documented requirements or software scope, poor development environment). *Predictable risks* are extrapolated from past project experience (e.g., staff turnover, poor communication with the customer, dilution of staff effort as ongoing maintenance requests are serviced). *Unpredictable risks* are the joker in the deck. They can and do occur, but they are extremely difficult to identify in advance.

---

1 A risk that is 100 percent probable is a constraint on the software project.

6.3 RISK IDENTIFICATION

Risk identification is a systematic attempt to specify threats to the project plan (estimates, schedule, resource loading, etc.). By identifying known and predictable risks, the project manager takes a first step toward avoiding them when possible and controlling them when necessary.

Although generic risks are important to consider, usually the product-specific risks cause the most headaches. Be certain to spend the time to identify as many product-specific risks as possible.

There are two distinct types of risks for each of the categories that have been presented in Section 6.2: generic risks and product-specific risks. *Generic risks* are a potential threat to every software project. *Product-specific risks* can be identified only by those with a clear understanding of the technology, the people, and the environment that is specific to the project at hand. To identify product-specific risks, the project plan and the software statement of scope are examined and an answer to the following question is developed: "What special characteristics of this product may threaten our project plan?"

One method for identifying risks is to create a *risk item checklist.* The checklist can be used for risk identification and focuses on some subset of known and predictable risks in the following generic subcategories:

Risk item checklist

- *Product size*—risks associated with the overall size of the software to be built or modified.
- *Business impact*—risks associated with constraints imposed by management or the marketplace.
- *Customer characteristics*—risks associated with the sophistication of the customer and the developer's ability to communicate with the customer in a timely manner.
- *Process definition*—risks associated with the degree to which the software process has been defined and is followed by the development organization.
- *Development environment*—risks associated with the availability and quality of the tools to be used to build the product.
- *Technology to be built*—risks associated with the complexity of the system to be built and the "newness" of the technology that is packaged by the system.
- *Staff size and experience*—risks associated with the overall technical and project experience of the software engineers who will do the work.

The risk item checklist can be organized in different ways. Questions relevant to each of the topics can be answered for each software project. The answers to these questions allow the planner to estimate the impact of risk. A different risk item checklist format simply lists characteristics that are relevant to each generic subcategory. Finally, a set of "risk components and drivers" [AFC88] are listed along with their probability

of occurrence. Drivers for performance, support, cost, and schedule are discussed in answer to later questions.

Quote:
"Risk management is project management for adults."
Tim Lister

A number of comprehensive checklists for software project risk have been proposed in the literature (e.g., [SEI93], [KAR96]). These provide useful insight into generic risks for software projects and should be used whenever risk analysis and management is instituted. However, a relatively short list of questions [KEI98] can be used to provide a preliminary indication of whether a project is "at risk."

6.3.1 Assessing Overall Project Risk

The following questions have derived from risk data obtained by surveying experienced software project managers in different part of the world [KEI98]. The questions are ordered by their relative importance to the success of a project.

? Is the software project we're working on at serious risk?

1. Have top software and customer managers formally committed to support the project?
2. Are end-users enthusiastically committed to the project and the system/product to be built?
3. Are requirements fully understood by the software engineering team and their customers?
4. Have customers been involved fully in the definition of requirements?
5. Do end-users have realistic expectations?
6. Is project scope stable?
7. Does the software engineering team have the right mix of skills?
8. Are project requirements stable?
9. Does the project team have experience with the technology to be implemented?
10. Is the number of people on the project team adequate to do the job?
11. Do all customer/user constituencies agree on the importance of the project and on the requirements for the system/product to be built?

WebRef
Risk Radar is a risk management database that helps project managers identify, rank, and communicate project risks. It can be found at **www.spmn.com/rsktrkr.html**

If any one of these questions is answered negatively, mitigation, monitoring, and management steps should be instituted without fail. The degree to which the project is at risk is directly proportional to the number of negative responses to these questions.

6.3.2 Risk Components and Drivers

The U.S. Air Force [AFC88] has written a pamphlet that contains excellent guidelines for software risk identification and abatement. The Air Force approach requires that the project manager identify the risk drivers that affect software risk components—

performance, cost, support, and schedule. In the context of this discussion, the risk components are defined in the following manner:

- *Performance risk*—the degree of uncertainty that the product will meet its requirements and be fit for its intended use.
- *Cost risk*—the degree of uncertainty that the project budget will be maintained.
- *Support risk*—the degree of uncertainty that the resultant software will be easy to correct, adapt, and enhance.
- *Schedule risk*—the degree of uncertainty that the project schedule will be maintained and that the product will be delivered on time.

The impact of each risk driver on the risk component is divided into one of four impact categories—negligible, marginal, critical, or catastrophic. Referring to Figure 6.1 [BOE89],

Components / Category		Performance	Support	Cost	Schedule
Catastrophic	1	Failure to meet the requirement would result in mission failure		Failure results in increased costs and schedule delays with expected values in excess of $500K	
	2	Significant degradation to nonachievement of technical performance	Nonresponsive or unsupportable software	Significant financial shortages, budget overrun likely	Unachievable IOC
Critical	1	Failure to meet the requirement would degrade system performance to a point where mission success is questionable		Failure results in operational delays and/or increased costs with expected value of $100K to $500K	
	2	Some reduction in technical performance	Minor delays in software modifications	Some shortage of financial resources, possible overruns	Possible slippage in IOC
Marginal	1	Failure to meet the requirement would result in degradation of secondary mission		Costs, impacts, and/or recoverable schedule slips with expected value of $1K to $100K	
	2	Minimal to small reduction in technical performance	Responsive software support	Sufficient financial resources	Realistic, achievable schedule
Negligible	1	Failure to meet the requirement would create inconvenience or nonoperational impact		Error results in minor cost and/or schedule impact with expected value of less than $1K	
	2	No reduction in technical performance	Easily supportable software	Possible budget underrun	Early achievable IOC

Note: (1) The potential consequence of undetected software errors or faults.
(2) The potential consequence if the desired outcome is not achieved.

FIGURE 6.1 Impact assessment [BOE89]

a characterization of the potential consequences of errors (rows labeled 1) or a failure to achieve a desired outcome (rows labeled 2) are described. The impact category is chosen based on the characterization that best fits the description in the table.

6.4 RISK PROJECTION

Risk projection, also called *risk estimation,* attempts to rate each risk in two ways—the likelihood or probability that the risk is real and the consequences of the problems associated with the risk, should it occur. The project planner, along with other managers and technical staff, performs four risk projection activities: (1) establish a scale that reflects the perceived likelihood of a risk, (2) delineate the consequences of the risk, (3) estimate the impact of the risk on the project and the product, and (4) note the overall accuracy of the risk projection so that there will be no misunderstandings.

6.4.1 Developing a Risk Table

A risk table provides a project manager with a simple technique for risk projection.[2] A sample risk table is illustrated in Figure 6.2.

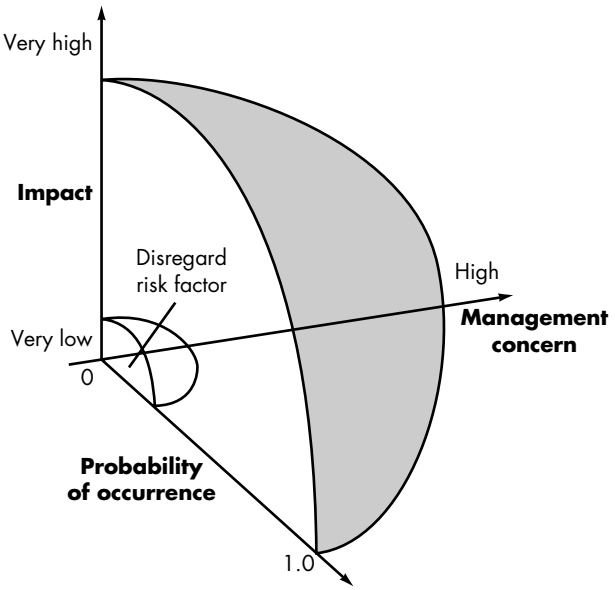
Risks	Category	Probability	Impact	RMMM
Size estimate may be significantly low	PS	60%	2	
Larger number of users than planned	PS	30%	3	
Less reuse than planned	PS	70%	2	
End-users resist system	BU	40%	3	
Delivery deadline will be tightened	BU	50%	2	
Funding will be lost	CU	40%	1	
Customer will change requirements	PS	80%	2	
Technology will not meet expectations	TE	30%	1	
Lack of training on tools	DE	80%	3	
Staff inexperienced	ST	30%	2	
Staff turnover will be high	ST	60%	2	
•				
•				
•				

Impact values:
1—catastrophic
2—critical
3—marginal
4—negligible

FIGURE 6.2 Sample risk table prior to sorting

2 The risk table should be implemented as a spreadsheet model. This enables easy manipulation and sorting of the entries.

FIGURE 6.3
Risk and management concern



Think hard about the software you're about to build and ask yourself, "What can go wrong?" Create your own list and ask other members of the software team to do the same.

A project team begins by listing all risks (no matter how remote) in the first column of the table. This can be accomplished with the help of the risk item checklists referenced in Section 6.3. Each risk is categorized in the second column (e.g., PS implies a project size risk, BU implies a business risk). The probability of occurrence of each risk is entered in the next column of the table. The probability value for each risk can be estimated by team members individually. Individual team members are polled in round-robin fashion until their assessment of risk probability begins to converge.

Next, the impact of each risk is assessed. Each risk component is assessed using the characterization presented in Figure 6.1, and an impact category is determined. The categories for each of the four risk components—performance, support, cost, and schedule—are averaged[3] to determine an overall impact value.

The risk table is sorted by probability and impact to rank risks.

Once the first four columns of the risk table have been completed, the table is sorted by probability and by impact. High-probability, high-impact risks percolate to the top of the table, and low-probability risks drop to the bottom. This accomplishes first-order risk prioritization.

The project manager studies the resultant sorted table and defines a cutoff line. The *cutoff line* (drawn horizontally at some point in the table) implies that only risks that lie above the line will be given further attention. Risks that fall below the line are re-evaluated to accomplish second-order prioritization. Referring to Figure 6.3, risk impact and probability have a distinct influence on management concern. A risk fac-

3 A weighted average can be used if one risk component has more significance for the project.

tor that has a high impact but a very low probability of occurrence should not absorb a significant amount of management time. However, high-impact risks with moderate to high probability and low-impact risks with high probability should be carried forward into the risk analysis steps that follow.

Quote:
"Failure to prepare is preparing to fail."
Ben Franklin

All risks that lie above the cutoff line must be managed. The column labeled RMMM contains a pointer into a *Risk Mitigation, Monitoring and Management Plan* or alternatively, a collection of risk information sheets developed for all risks that lie above the cutoff. The RMMM plan and risk information sheets are discussed in Sections 6.5 and 6.6.

Risk probability can be determined by making individual estimates and then developing a single consensus value. Although that approach is workable, more sophisticated techniques for determining risk probability have been developed [AFC88]. Risk drivers can be assessed on a qualitative probability scale that has the following values: impossible, improbable, probable, and frequent. Mathematical probability can then be associated with each qualitative value (e.g., a probability of 0.7 to 1.0 implies a highly probable risk).

6.4.2 Assessing Risk Impact

Three factors affect the consequences that are likely if a risk does occur: its nature, its scope, and its timing. The *nature* of the risk indicates the problems that are likely if it occurs. For example, a poorly defined external interface to customer hardware (a technical risk) will preclude early design and testing and will likely lead to system integration problems late in a project. The *scope* of a risk combines the severity (just how serious is it?) with its overall distribution (how much of the project will be affected or how many customers are harmed?). Finally, the *timing* of a risk considers when and for how long the impact will be felt. In most cases, a project manager might want the "bad news" to occur as soon as possible, but in some cases, the longer the delay, the better.

Returning once more to the risk analysis approach proposed by the U.S. Air Force [AFC88], the following steps are recommended to determine the overall consequences of a risk:

? How do we assess the consequences of a risk?

1. Determine the average probability of occurrence value for each risk component.
2. Using Figure 6.1, determine the impact for each component based on the criteria shown.
3. Complete the risk table and analyze the results as described in the preceding sections.

The overall *risk exposure,* RE, is determined using the following relationship [HAL98]:

$$RE = P \times C$$

where P is the probability of occurrence for a risk, and C is the the cost to the project should the risk occur.

For example, assume that the software team defines a project risk in the following manner:

Risk identification. Only 70 percent of the software components scheduled for reuse will, in fact, be integrated into the application. The remaining functionality will have to be custom developed.

Risk probability. 80% (likely).

Risk impact. 60 reusable software components were planned. If only 70 percent can be used, 18 components would have to be developed from scratch (in addition to other custom software that has been scheduled for development). Since the average component is 100 LOC and local data indicate that the software engineering cost for each LOC is $14.00, the overall cost (impact) to develop the components would be 18 x 100 x 14 = $25,200.

Risk exposure. *RE* = 0.80 x 25,200 ~ $20,200.

Compare RE for all risks to the cost estimate for the project. If RE is greater than 50 percent of project cost, the viability of the project must be evaluated.

Risk exposure can be computed for each risk in the risk table, once an estimate of the cost of the risk is made. The total risk exposure for all risks (above the cutoff in the risk table) can provide a means for adjusting the final cost estimate for a project. It can also be used to predict the probable increase in staff resources required at various points during the project schedule.

The risk projection and analysis techniques described in Sections 6.4.1 and 6.4.2 are applied iteratively as the software project proceeds. The project team should revisit the risk table at regular intervals, re-evaluating each risk to determine when new circumstances cause its probability and impact to change. As a consequence of this activity, it may be necessary to add new risks to the table, remove some risks that are no longer relevant, and change the relative positions of still others.

6.4.3 Risk Assessment

At this point in the risk management process, we have established a set of triplets of the form [CHA89]:

$$[r_i, l_i, x_i]$$

where r_i is risk, l_i is the likelihood (probability) of the risk, and x_i is the impact of the risk. During risk assessment, we further examine the accuracy of the estimates that were made during risk projection, attempt to rank the risks that have been uncovered, and begin thinking about ways to control and/or avert risks that are likely to occur.

For assessment to be useful, a *risk referent level* [CHA89] must be defined. For most software projects, the risk components discussed earlier—performance, cost, support, and schedule—also represent risk referent levels. That is, there is a level for per-

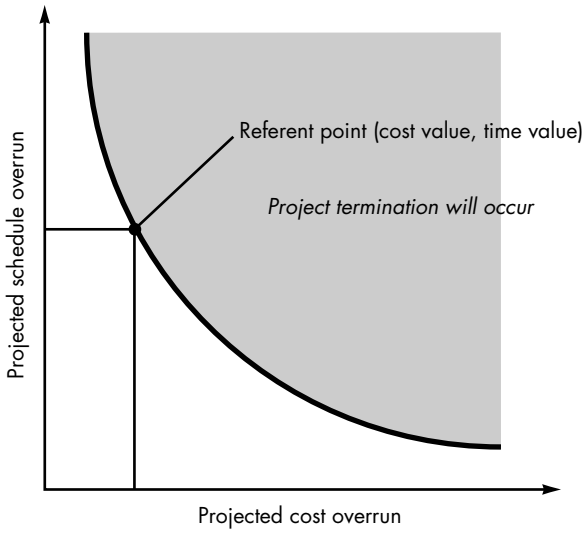


FIGURE 6.4
Risk referent level

KEY POINT
The risk referent level establishes your tolerance for pain. Once risk exposure exceeds the referent level, the project may be terminated.

formance degradation, cost overrun, support difficulty, or schedule slippage (or any combination of the four) that will cause the project to be terminated. If a combination of risks create problems that cause one or more of these referent levels to be exceeded, work will stop. In the context of software risk analysis, a risk referent level has a single point, called the *referent point* or *break point,* at which the decision to proceed with the project or terminate it (problems are just too great) are equally weighted. Figure 6.4 represents this situation graphically.

In reality, the referent level can rarely be represented as a smooth line on a graph. In most cases it is a region in which there are areas of uncertainty; that is, attempting to predict a management decision based on the combination of referent values is often impossible. Therefore, during risk assessment, we perform the following steps:

1. Define the risk referent levels for the project.
2. Attempt to develop a relationship between each (r_i, l_i, x_i) and each of the referent levels.
3. Predict the set of referent points that define a region of termination, bounded by a curve or areas of uncertainty.
4. Try to predict how compound combinations of risks will affect a referent level.

A detailed discussion of risk referent level is best left to books that are dedicated to risk analysis (e.g., [CHA89], [ROW88]).

6.5 RISK REFINEMENT

During early stages of project planning, a risk may be stated quite generally. As time passes and more is learned about the project and the risk, it may be possible to refine the risk into a set of more detailed risks, each somewhat easier to mitigate, monitor, and manage.

What is a good way to describe a risk?

One way to do this is to represent the risk in *condition-transition-consequence* (CTC) format [GLU94]. That is, the risk is stated in the following form:

Given that <condition> then there is concern that (possibly) <consequence>.

Using the CTC format for the reuse risk noted in Section 6.4.2, we can write:

Given that all reusable software components must conform to specific design standards and that some do not conform, then there is concern that (possibly) only 70 percent of the planned reusable modules may actually be integrated into the as-built system, resulting in the need to custom engineer the remaining 30 percent of components.

This general condition can be refined in the following manner:

Subcondition 1. Certain reusable components were developed by a third party with no knowledge of internal design standards.

Subcondition 2. The design standard for component interfaces has not been solidified and may not conform to certain existing reusable components.

Subcondition 3. Certain reusable components have been implemented in a language that is not supported on the target environment.

The consequences associated with these refined subconditions remains the same (i.e., 30 percent of software components must be customer engineered), but the refinement helps to isolate the underlying risks and might lead to easier analysis and response.

6.6 RISK MITIGATION, MONITORING, AND MANAGEMENT

All of the risk analysis activities presented to this point have a single goal—to assist the project team in developing a strategy for dealing with risk. An effective strategy must consider three issues:

Quote:

"If I take so many precautions, it is because I leave nothing to chance."

Napolean

- risk avoidance
- risk monitoring
- risk management and contingency planning

If a software team adopts a proactive approach to risk, avoidance is always the best strategy. This is achieved by developing a plan for *risk mitigation.* For example, assume that high staff turnover is noted as a project risk, r_1. Based on past history and man-

agement intuition, the likelihood, l_1, of high turnover is estimated to be 0.70 (70 percent, rather high) and the impact, x_1, is projected at level 2. That is, high turnover will have a critical impact on project cost and schedule.

WebRef

An excellent FAQ on risk management can be obtained at **www.sei.cmu.edu/organization/programs/sepm/risk/risk.faq.html**

To mitigate this risk, project management must develop a strategy for reducing turnover. Among the possible steps to be taken are

- Meet with current staff to determine causes for turnover (e.g., poor working conditions, low pay, competitive job market).
- Mitigate those causes that are under our control before the project starts.
- Once the project commences, assume turnover will occur and develop techniques to ensure continuity when people leave.
- Organize project teams so that information about each development activity is widely dispersed.
- Define documentation standards and establish mechanisms to be sure that documents are developed in a timely manner.
- Conduct peer reviews of all work (so that more than one person is "up to speed").
- Assign a backup staff member for every critical technologist.

As the project proceeds, risk monitoring activities commence. The project manager monitors factors that may provide an indication of whether the risk is becoming more or less likely. In the case of high staff turnover, the following factors can be monitored:

Quote:

"We are ready for an unforseen event that may or may not occur."

Dan Quayle

- General attitude of team members based on project pressures.
- The degree to which the team has jelled.
- Interpersonal relationships among team members.
- Potential problems with compensation and benefits.
- The availability of jobs within the company and outside it.

In addition to monitoring these factors, the project manager should monitor the effectiveness of risk mitigation steps. For example, a risk mitigation step noted here called for the definition of documentation standards and mechanisms to be sure that documents are developed in a timely manner. This is one mechanism for ensuring continuity, should a critical individual leave the project. The project manager should monitor documents carefully to ensure that each can stand on its own and that each imparts information that would be necessary if a newcomer were forced to join the software team somewhere in the middle of the project.

Risk management and contingency planning assumes that mitigation efforts have failed and that the risk has become a reality. Continuing the example, the project is

well underway and a number of people announce that they will be leaving. If the mitigation strategy has been followed, backup is available, information is documented, and knowledge has been dispersed across the team. In addition, the project manager may temporarily refocus resources (and readjust the project schedule) to those functions that are fully staffed, enabling newcomers who must be added to the team to "get up to speed." Those individuals who are leaving are asked to stop all work and spend their last weeks in "knowledge transfer mode." This might include video-based knowledge capture, the development of "commentary documents," and/or meeting with other team members who will remain on the project.

If RE for a specific risk is less than the cost of risk mitigation, don't try to mitigate the risk but continue to monitor it.

It is important to note that RMMM steps incur additional project cost. For example, spending the time to "backup" every critical technologist costs money. Part of risk management, therefore, is to evaluate when the benefits accrued by the RMMM steps are outweighed by the costs associated with implementing them. In essence, the project planner performs a classic cost/benefit analysis. If risk aversion steps for high turnover will increase both project cost and duration by an estimated 15 percent, but the predominant cost factor is "backup," management may decide not to implement this step. On the other hand, if the risk aversion steps are projected to increase costs by 5 percent and duration by only 3 percent management will likely put all into place.

For a large project, 30 or 40 risks may identified. If between three and seven risk management steps are identified for each, risk management may become a project in itself! For this reason, we adapt the Pareto 80–20 rule to software risk. Experience indicates that 80 percent of the overall project risk (i.e., 80 percent of the potential for project failure) can be accounted for by only 20 percent of the identified risks. The work performed during earlier risk analysis steps will help the planner to determine which of the risks reside in that 20 percent (e.g., risks that lead to the highest risk exposure). For this reason, some of the risks identified, assessed, and projected may not make it into the RMMM plan—they don't fall into the critical 20 percent (the risks with highest project priority).

6.7 SAFETY RISKS AND HAZARDS

Risk is not limited to the software project itself. Risks can occur after the software has been successfully developed and delivered to the customer. These risks are typically associated with the consequences of software failure in the field.

In the early days of computing, there was reluctance to use computers (and software) to control safety critical processes such as nuclear reactors, aircraft flight control, weapons systems, and large-scale industrial processes. Although the probability of failure of a well-engineered system was small, an undetected fault in a computer-based control or monitoring system could result in enormous economic damage or, worse, significant human injury or loss of life. But the cost and functional benefits of

computer-based control and monitoring far outweigh the risk. Today, computer hardware and software are used regularly to control safety critical systems.

When software is used as part of a control system, complexity can increase by an order of magnitude or more. Subtle design faults induced by human error—something that can be uncovered and eliminated in hardware-based conventional control—become much more difficult to uncover when software is used.

Software safety and *hazard analysis* [LEV95] are software quality assurance activities (Chapter 8) that focus on the identification and assessment of potential hazards that may affect software negatively and cause an entire system to fail. If hazards can be identified early in the software engineering process, software design features can be specified that will either eliminate or control potential hazards.

WebRef

A voluminous database containing all entries from the ACM's Forum on Risks to the Public can be found at **catless.ncl.ac.uk/Risks/search.html**

6.8 THE RMMM PLAN

A risk management strategy can be included in the software project plan or the risk management steps can be organized into a separate *Risk Mitigation, Monitoring and Management Plan.* The RMMM plan documents all work performed as part of risk analysis and is used by the project manager as part of the overall project plan.

RMMM Plan

Some software teams do not develop a formal RMMM document. Rather, each risk is documented individually using a *risk information sheet* (RIS) [WIL97]. In most cases, the RIS is maintained using a database system, so that creation and information entry, priority ordering, searches, and other analysis may be accomplished easily. The format of the RIS is illustrated in Figure 6.5.

Once RMMM has been documented and the project has begun, risk mitigation and monitoring steps commence. As we have already discussed, risk mitigation is a problem avoidance activity. Risk monitoring is a project tracking activity with three primary objectives: (1) to assess whether predicted risks do, in fact, occur; (2) to ensure that risk aversion steps defined for the risk are being properly applied; and (3) to collect information that can be used for future risk analysis. In many cases, the problems that occur during a project can be traced to more than one risk. Another job of risk monitoring is to attempt to allocate *origin* (what risk(s) caused which problems throughout the project).

6.9 SUMMARY

Whenever a lot is riding on a software project, common sense dictates risk analysis. And yet, most software project managers do it informally and superficially, if they do it at all. The time spent identifying, analyzing, and managing risk pays itself back in many ways: less upheaval during the project, a greater ability to track and control a project, and the confidence that comes with planning for problems before they occur.

FIGURE 6.5
Risk information sheet [WIL97]

Risk information sheet			
Risk ID: P02-4-32	Date: 5/9/02	Prob: 80%	Impact: high
Description: Only 70 percent of the software components scheduled for reuse will, in fact, be integrated into the application. The remaining functionality will have to be custom developed.			
Refinement/context: Subcondition 1: Certain reusable components were developed by a third party with no knowledge of internal design standards. Subcondition 2: The design standard for component interfaces has not been solidified and may not conform to certain existing reusable components. Subcondition 3: Certain reusable components have been implemented in a language that is not supported on the target environment.			
Mitigation/monitoring: 1. Contact third party to determine conformance with design standards. 2. Press for interface standards completion; consider component structure when deciding on interface protocol. 3. Check to determine number of components in subcondition 3 category; check to determine if language support can be acquired.			
Management/contingency plan/trigger: *RE* computed to be $20,200. Allocate this amount within project contingency cost. Develop revised schedule assuming that 18 additional components will have to be custom built; allocate staff accordingly. Trigger: Mitigation steps unproductive as of 7/1/02			
Current status: 5/12/02: Mitigation steps initiated.			
Originator: D. Gagne		Assigned: B. Laster	

Risk analysis can absorb a significant amount of project planning effort. Identification, projection, assessment, management, and monitoring all take time. But the effort is worth it. To quote Sun Tzu, a Chinese general who lived 2500 years ago, "If you know the enemy and know yourself, you need not fear the result of a hundred battles." For the software project manager, the enemy is risk.

REFERENCES

[AFC88] *Software Risk Abatement,* AFCS/AFLC Pamphlet 800-45, U.S. Air Force, September 30, 1988.

[BOE89] Boehm, B.W., *Software Risk Management,* IEEE Computer Society Press, 1989.

[CHA89] Charette, R.N., *Software Engineering Risk Analysis and Management,* McGraw-Hill/Intertext, 1989.

[CHA92] Charette, R.N., "Building Bridges over Intelligent Rivers," *American Programmer,* vol. 5, no. 7, September, 1992, pp. 2–9.
[DRU75] Drucker, P., *Management,* W. H. Heinemann, 1975.
[GIL88] Gilb, T., *Principles of Software Engineering Management,* Addison-Wesley, 1988.
[GLU94] Gluch, D.P., "A Construct for Describing Software Development Risks," CMU/SEI-94-TR-14, Software Engineering Institute, 1994.
[HAL98] Hall, E.M., *Managing Risk: Methods for Software Systems Development,* Addison-Wesley, 1998.
[HIG95] Higuera, R.P., "Team Risk Management," *CrossTalk,* U.S. Dept. of Defense, January 1995, p. 2–4.
[KAR96] Karolak, D.W., *Software Engineering Risk Management,* IEEE Computer Society Press, 1996.
[KEI98] Keil, M., et al., "A Framework for Identifying Software Project Risks," *CACM,* vol. 41, no. 11, November 1998, pp. 76–83.
[LEV95] Leveson, N.G., *Safeware: System Safety and Computers,* Addison-Wesley, 1995.
[ROW88] Rowe, W.D., *An Anatomy of Risk,* Robert E. Krieger Publishing Co., 1988.
[SEI93] "Taxonomy-Based Risk Identification," Software Engineering Institute, CMU/SEI-93-TR-6, 1993.
[THO92] Thomsett, R., "The Indiana Jones School of Risk Management," *American Programmer,* vol. 5, no. 7, September 1992, pp. 10–18.
[WIL97] Williams, R.C, J.A. Walker, and A.J. Dorofee, "Putting Risk Management into Practice," *IEEE Software,* May 1997, pp. 75–81.

PROBLEMS AND POINTS TO PONDER

6.1. Provide five examples from other fields that illustrate the problems associated with a reactive risk strategy.

6.2. Describe the difference between "known risks" and "predictable risks."

6.3. Add three additional questions or topics to each of the risk item checklists presented at the SEPA Web site.

6.4. You've been asked to build software to support a low-cost video editing system. The system accepts videotape as input, stores the video on disk, and then allows the user to do a wide range of edits to the digitized video. The result can then be output to tape. Do a small amount of research on systems of this type and then make a list of technology risks that you would face as you begin a project of this type.

6.5. You're the project manager for a major software company. You've been asked to lead a team that's developing "next generation" word-processing software (see Section 3.4.2 for a brief description). Create a risk table for the project.

6.6. Describe the difference between risk components and risk drivers.

6.7. Develop a risk mitigation strategy and specific risk mitigation activities for three of the risks noted in Figure 6.2.

6.8. Develop a risk monitoring strategy and specific risk monitoring activities for three of the risks noted in Figure 6.2. Be sure to identify the factors that you'll be monitoring to determine whether the risk is becoming more or less likely.

6.9. Develop a risk management strategy and specific risk management activities for three of the risks noted in Figure 6.2.

6.10. Attempt to refine three of the risks noted in Figure 6.2 and then create risk information sheets for each.

6.11. Represent three of the risks noted in Figure 6.2 using a CTC format.

6.12. Recompute the risk exposure discussed in Section 6.4.2 when cost/LOC is $16 and the probability is 60 percent.

6.13. Can you think of a situation in which a high-probability, high-impact risk would not be considered as part of your RMMM plan?

6.14. Referring the the risk referent shown on Figure 6.4, would the curve always have the symmetric arc shown or would there be situations in which the curve would be more distorted. If so, suggest a scenario in which this might happen.

6.15. Do some research on software safety issues and write a brief paper on the subject. Do a Web search to get current information.

6.16. Describe five software application areas in which software safety and hazard analysis would be a major concern.

FURTHER READINGS AND INFORMATION SOURCES

The software risk management literature has expanded significantly in recent years. Hall [HAL98] presents one of the more thorough treatments of the subject. Karolak [KAR96] has written a guidebook that introduces an easy-to-use risk analysis model with worthwhile checklists and questionnaires. A useful snapshot of risk assessment has been written by Grey (*Practical Risk Assessment for Project Management,* Wiley, 1995). His abbreviated treatment provides a good introduction to the subject. Additional books worth examining include

Chapman, C.B. and S. Ward, *Project Risk Management: Processes, Techniques and Insights,* Wiley, 1997.

Schuyler, J.R., *Decision Analysis in Projects,* Project Management Institute Publications, 1997.

Wideman, R.M. (editor), *Project & Program Risk Management: A Guide to Managing Project Risks and Opportunities,* Project Management Institute Publications, 1998.

Capers Jones (*Assessment and Control of Software Risks,* Prentice-Hall, 1994) presents a detailed discussion of software risks that includes data collected from hundreds of software projects. Jones defines 60 risk factors that can affect the outcome of software projects. Boehm [BOE89] suggests excellent questionnaire and checklist formats that can prove invaluable in identifying risk. Charette [CHA89] presents a detailed treatment of the mechanics of risk analysis, calling on probability theory and statistical techniques to analyze risks. In a companion volume, Charette (*Application Strategies for Risk Analysis,* McGraw-Hill, 1990) discusses risk in the context of both system and software engineering and suggests pragmatic strategies for risk management. Gilb (*Principles of Software Engineering Management,* Addison-Wesley, 1988) presents a set of "principles" (which are often amusing and sometimes profound) that can serve as a worthwhile guide for risk management.

The March 1995 issue of *American Programmer,* the May 1997 issue of *IEEE Software,* and the June 1998 issue of the *Cutter IT Journal* all are dedicated to risk management.

The Software Engineering Institute has published many detailed reports and guidebooks on risk analysis and management. The Air Force Systems Command pamphlet AFSCP 800-45 [AFC88] describes risk identification and reduction techniques. Every issue of the *ACM Software Engineering Notes* has a section entitled "Risks to the Public" (editor, P.G. Neumann). If you want the latest and best software horror stories, this is the place to go.

A wide variety of information sources on risk analysis and management is available on the Internet. An up-to-date list of World Wide Web references that are relevant to risk can be found at the SEPA Web site:

http://www.mhhe.com/engcs/compsci/pressman/resources/risk.mhtml

CHAPTER

7 PROJECT SCHEDULING AND TRACKING

KEY CONCEPTS

adaptation criteria 174
critical path. 181
earned value . . . 186
error tracking. . . 187
lateness 166
people and effort 170
project plan 189
project tracking. 185
scheduling principles 168
task network. . . 180
task set 172
timeline chart. . . 182
work breakdown structure. 181

In the late 1960s, a bright-eyed young engineer was chosen to "write" a computer program for an automated manufacturing application. The reason for his selection was simple. He was the only person in his technical group who had attended a computer programming seminar. He knew the ins and outs of assembly language and FORTRAN but nothing about software engineering and even less about project scheduling and tracking.

His boss gave him the appropriate manuals and a verbal description of what had to be done. He was informed that the project must be completed in two months.

He read the manuals, considered his approach, and began writing code. After two weeks, the boss called him into his office and asked how things were going.

"Really great," said the young engineer with youthful enthusiasm, "This was much simpler than I thought. I'm probably close to 75 percent finished."

The boss smiled. "That's really terrific," he said, encouraging the young engineer to keep up the good work. They planned to meet again in a week's time.

A week later the boss called the engineer into his office and asked, "Where are we?"

QUICK LOOK

What is it? You've selected an appropriate process model, you've identified the software engineering tasks that have to be performed, you estimated the amount of work and the number of people, you know the deadline, you've even considered the risks. Now it's time to connect the dots. That is, you have to create a network of software engineering tasks that will enable you to get the job done on time. Once the network is created, you have to assign responsibility for each task, make sure it gets done, and adapt the network as risks become reality. In a nutshell, that's software project scheduling and tracking.

Who does it? At the project level, software proj-ect managers using information solicited from software engineers. At an individual level, software engineers themselves.

Why is it important? In order to build a complex system, many software engineering tasks occur in parallel, and the result of work performed during one task may have a profound effect on work to be conducted in another task. These interdependencies are very difficult to understand without a schedule. It's also virtually impossible to assess progress on a moderate or large software project without a detailed schedule.

What are the steps? The software engineering tasks dictated by the software process model are refined for the functionality to be built. Effort and duration are allocated to each task and a task network (also called an "activity network") is

QUICK LOOK

created in a manner that enables the software team to meet the delivery deadline established.

What is the work product? The project schedule and related information are produced.

How do I ensure that I've done it right? Proper scheduling requires that (1) all tasks appear in the network, (2) effort and timing are intelligently allocated to each task, (3) interdependencies between tasks are properly indicated, (4) resources are allocated for the work to be done, and (5) closely spaced milestones are provided so that progress can be tracked.

"Everything's going well," said the youngster, "but I've run into a few small snags. I'll get them ironed out and be back on track soon."

"How does the deadline look?" the boss asked.

"No problem," said the engineer. "I'm close to 90 percent complete."

If you've been working in the software world for more than a few years, you can finish the story. It'll come as no surprise that the young engineer[1] stayed 90 percent complete for the entire project duration and finished (with the help of others) only one month late.

This story has been repeated tens of thousands of times by software developers during the past three decades. The big question is why?

7.1 BASIC CONCEPTS

Although there are many reasons why software is delivered late, most can be traced to one or more of the following root causes:

Quote:

"Excessive or irrational schedules are probably the single most destructive influence in all of software."

Capers Jones

- An unrealistic deadline established by someone outside the software development group and forced on managers and practitioner's within the group.
- Changing customer requirements that are not reflected in schedule changes.
- An honest underestimate of the amount of effort and/or the number of resources that will be required to do the job.
- Predictable and/or unpredictable risks that were not considered when the project commenced.
- Technical difficulties that could not have been foreseen in advance.
- Human difficulties that could not have been foreseen in advance.
- Miscommunication among project staff that results in delays.
- A failure by project management to recognize that the project is falling behind schedule and a lack of action to correct the problem.

Aggressive (read "unrealistic") deadlines are a fact of life in the software business. Sometimes such deadlines are demanded for reasons that are legitimate, from the

1 If you're wondering whether this story is autobiographical, it is!

point of view of the person who sets the deadline. But common sense says that legitimacy must also be perceived by the people doing the work.

7.1.1 Comments on "Lateness"

Napoleon once said: "Any commander in chief who undertakes to carry out a plan which he considers defective is at fault; he must put forth his reasons, insist on the plan being changed, and finally tender his resignation rather than be the instrument of his army's downfall." These are strong words that many software project managers should ponder.

Quote:

"I love deadlines. I like the whooshing sound they make as they fly by."

Douglas Adams

The estimation and risk analysis activities discussed in Chapters 5 and 6, and the scheduling techniques described in this chapter are often implemented under the constraint of a defined deadline. If best estimates indicate that the deadline is unrealistic, a competent project manager should "protect his or her team from undue [schedule] pressure . . . [and] reflect the pressure back to its originators" [PAG85].

To illustrate, assume that a software development group has been asked to build a real-time controller for a medical diagnostic instrument that is to be introduced to the market in nine months. After careful estimation and risk analysis, the software project manager comes to the conclusion that the software, as requested, will require 14 calendar months to create with available staff. How does the project manager proceed?

It is unrealistic to march into the customer's office (in this case the likely customer is marketing/sales) and demand that the delivery date be changed. External market pressures have dictated the date, and the product must be released. It is equally foolhardy to refuse to undertake the work (from a career standpoint). So, what to do?

The following steps are recommended in this situation:

? What should we do when management demands that we make a deadline that is impossible?

1. Perform a detailed estimate using historical data from past projects. Determine the estimated effort and duration for the project.
2. Using an incremental process model (Chapter 2), develop a software engineering strategy that will deliver critical functionality by the imposed deadline, but delay other functionality until later. Document the plan.
3. Meet with the customer and (using the detailed estimate), explain why the imposed deadline is unrealistic. Be certain to note that all estimates are based on performance on past projects. Also be certain to indicate the percent improvement that would be required to achieve the deadline as it currently exists.[2] The following comment is appropriate:

   "I think we may have a problem with the delivery date for the XYZ controller software. I've given each of you an abbreviated breakdown of production

2 If the percent of improvement is 10 to 25 percent, it may actually be possible to get the job done. But, more likely, the percent of improvement in team performance must be greater than 50 percent. This is an unrealistic expectation.

rates for past projects and an estimate that we've done a number of different ways. You'll note that I've assumed a 20 percent improvement in past production rates, but we still get a delivery date that's 14 calendar months rather than 9 months away."

4. Offer the incremental development strategy as an alternative:

   "We have a few options, and I'd like you to make a decision based on them. First, we can increase the budget and bring on additional resources so that we'll have a shot at getting this job done in nine months. But understand that this will increase risk of poor quality due to the tight timeline.[3] Second, we can remove a number of the software functions and capabilities that you're requesting. This will make the preliminary version of the product somewhat less functional, but we can announce all functionality and then deliver over the 14 month period. Third, we can dispense with reality and wish the project complete in nine months. We'll wind up with nothing that can be delivered to a customer. The third option, I hope you'll agree, is unacceptable. Past history and our best estimates say that it is unrealistic and a recipe for disaster."

XRef

Incremental process models are described in Chapter 2.

There will be some grumbling, but if solid estimates based on good historical data are presented, it's likely that negotiated versions of option 1 or 2 will be chosen. The unrealistic deadline evaporates.

7.1.2 Basic Principles

Fred Brooks, the well-known author of *The Mythical Man-Month* [BRO95], 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.[4] If these "critical" tasks fall behind schedule, the completion date of the entire project is put into jeopardy.

The tasks required to achieve the project manager's objective should not be performed manually. There are many excellent project scheduling tools. Use them.

The project manager's 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, the manager must have a schedule that has been defined at a degree of resolution that enables the manager to monitor progress and control the project.

Software project scheduling is an activity that distributes estimated effort across the planned project duration by allocating the effort to specific software engineering tasks.

---

3 You might also add that adding more people does not reduce calendar time proportionally.

4 The critical path will be discussed in greater detail later in this chapter.

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 software engineering 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 tasks (required to accomplish an activity) are identified and scheduled.

Quote:

"Overly optimistic scheduling doesn't result in shorter actual schedules, it results in longer ones."

Steve McConnell

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.

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 decomposed (Chapter 3).

KEY POINT

When you develop a schedule, compartmentalize the work, represent task interdependencies, allocate effort and time to each task, define responsibilities for the work to be done, and define outcomes and milestones.

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 staff members. As time allocation occurs, the project manager must ensure that no more than the allocated number of people have been scheduled at any given time. For example, consider a project that has three assigned staff members (e.g., 3 person-days are available per day of assigned effort[5]). 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.

---

5 In reality, less than three person-days are available because of unrelated meetings, sickness, vacation, and a variety of other reasons. For our purposes, however, we assume 100 percent availability.

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 module) 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 8) and has been approved.

Each of these principles is applied as the project schedule evolves.

7.2 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 ten person-year effort with one person working for ten years!)

There is a common myth (discussed in Chapter 1) that is still believed by many managers who are responsible for software development effort: "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.

If you must add people to a late project, be certain that you've assigned them work that is highly compartmentalized.

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.

7.2.1 An Example

Consider four software engineers, each capable of producing 5000 LOC/year when working on an individual project. When these four engineers are placed on a team project, six potential communication paths are possible. Each communication path requires time that could otherwise be spent developing software. We shall assume that team productivity (when measured in LOC) will be reduced by 250 LOC/year for each communication path, due to the overhead associated with communication. Therefore, team productivity is 20,000 − (250 x 6) = 18,500 LOC/year—7.5 percent less than what we might expect.[6]

6 It is possible to pose a counterargument: Communication, if it is effective, can enhance the quality of the work being performed, thereby reducing the amount of rework and increasing the individual productivity of team members. The jury is still out!

The relationship between the number of people working on a software project and overall productivity is not linear.

The one-year project on which the team is working falls behind schedule, and with two months remaining, two additional people are added to the team. The number of communication paths escalates to 14. The productivity input of the new staff is the equivalent of 840 x 2 = 1680 LOC for the two months remaining before delivery. Team productivity now is 20,000 + 1680 − (250 x 14) = 18,180 LOC/year.

Although the example is a gross oversimplification of real-world circumstances, it does illustrate another key point: The relationship between the number of people working on a software project and overall productivity is not linear.

Based on the people/work relationship, are teams counterproductive? The answer is an emphatic "no," if communication improves software quality. In fact, formal technical reviews (see Chapter 8) conducted by software teams can lead to better analysis and design, and more important, can reduce the number of errors that go undetected until testing (thereby reducing testing effort). Hence, productivity and quality, when measured by time to project completion and customer satisfaction, can actually improve.

7.2.2. An Empirical Relationship

Recalling the software equation [PUT92] that was introduced in Chapter 5, we can demonstrate 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:

$$L = P \times E^{1/3} t^{4/3}$$

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 2,000 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:*

$$E = L^3 / (P^3 t^4) \qquad (7\text{-}1)$$

As the deadline becomes tighter and tighter, you reach a point at which the work cannot be completed on schedule, regardless of the number of people doing the work. Face reality and define a new delivery date.

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 (7-1) yields:

$E = L^3 / (P^3 t^4) \sim 3.8$ person-years.

This implies that, by extending the end-date 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.

7.2.3 Effort Distribution

? How much effort should be expended on each of the major software engineering tasks?

Each of the software project estimation techniques discussed in Chapter 5 leads to estimates of work units (e.g., person-months) required to complete software development. A recommended distribution of effort across the definition and development phases is often referred to as the *40–20–40 rule.*[7] Forty percent of all effort is allocated to front-end analysis and design. A similar percentage is applied to back-end testing. You can correctly infer that coding (20 percent of effort) is de-emphasized.

This effort distribution should be used as a guideline only. The characteristics of each project must dictate the distribution of effort. Work expended on project planning rarely accounts for more than 2–3 percent of effort, unless the plan commits an organization to large expenditures with high risk. Requirements analysis may comprise 10–25 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 15–20 percent of overall effort can be achieved. Testing and subsequent debugging can account for 30–40 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.

7.3 DEFINING A TASK SET FOR THE SOFTWARE PROJECT

A number of different process models were described in Chapter 2. These models offer different paradigms for software development. Regardless of whether a software team chooses a linear sequential paradigm, an iterative paradigm, an evolutionary paradigm, a concurrent paradigm or some permutation, the process model is populated by a set of tasks that enable a software team to define, develop, and ultimately support computer software.

No single set of tasks is appropriate for all projects. The set of tasks that would be appropriate for a large, complex system would likely be perceived as overkill for a small, relatively simple software product. Therefore, an effective software process

7 Today, more than 40 percent of all project effort is often recommended for analysis and design tasks for large software development projects. Hence, the name *40–20–40* no longer applies in a strict sense.

should define a collection of task sets, each designed to meet the needs of different types of projects.

KEY POINT
A "task set" is a collection of software engineering tasks, milestones, and deliverables.

A *task set* is a collection of software engineering work tasks, milestones, and deliverables that must be accomplished to complete a particular project. The task set to be chosen must provide enough discipline to achieve high software quality. But, at the same time, it must not burden the project team with unnecessary work.

Task sets are designed to accommodate different types of projects and different degrees of rigor. Although it is difficult to develop a comprehensive taxonomy of software project types, most software organizations encounter the following projects:

1. *Concept development projects* that are initiated to explore some new business concept or application of some new technology.
2. *New application development projects* that are undertaken as a consequence of a specific customer request.
3. *Application enhancement projects* that occur when existing software undergoes major modifications to function, performance, or interfaces that are observable by the end-user.
4. *Application maintenance projects* that correct, adapt, or extend existing software in ways that may not be immediately obvious to the end-user.
5. *Reengineering projects* that are undertaken with the intent of rebuilding an existing (legacy) system in whole or in part.

Even within a single project type, many factors influence the task set to be chosen. When taken in combination, these factors provide an indication of the degree of rigor with which the software process should be applied.

7.3.1 Degree of Rigor

KEY POINT
The task set will grow in size and complexity as the degree of rigor grows.

Even for a project of a particular type, the *degree of rigor* with which the software process is applied may vary significantly. The degree of rigor is a function of many project characteristics. As an example, small, non-business-critical projects can generally be addressed with somewhat less rigor than large, complex business-critical applications. It should be noted, however, that all projects must be conducted in a manner that results in timely, high-quality deliverables. Four different degrees of rigor can be defined:

Casual. All process framework activities (Chapter 2) are applied, but only a minimum task set is required. In general, umbrella tasks will be minimized and documentation requirements will be reduced. All basic principles of software engineering are still applicable.

Structured. The process framework will be applied for this project. Framework activities and related tasks appropriate to the project type will be applied and umbrella activities necessary to ensure high quality will be

applied. SQA, SCM, documentation, and measurement tasks will be conducted in a streamlined manner.

Strict. The full process will be applied for this project with a degree of discipline that will ensure high quality. All umbrella activities will be applied and robust work products will be produced.

Quick reaction. The process framework will be applied for this project, but because of an emergency situation[8] only those tasks essential to maintaining good quality will be applied. "Back-filling" (i.e., developing a complete set of documentation, conducting additional reviews) will be accomplished after the application/product is delivered to the customer.

If everything is an emergency, there's something wrong with your software process or with the people who manage the business or both.

The project manager must develop a systematic approach for selecting the degree of rigor that is appropriate for a particular project. To accomplish this, project adaptation criteria are defined and a task set selector value is computed.

7.3.2 Defining Adaptation Criteria

Adaptation criteria are used to determine the recommended degree of rigor with which the software process should be applied on a project. Eleven adaptation criteria [PRE99] are defined for software projects:

- Size of the project
- Number of potential users
- Mission criticality
- Application longevity
- Stability of requirements
- Ease of customer/developer communication
- Maturity of applicable technology
- Performance constraints
- Embedded and nonembedded characteristics
- Project staff
- Reengineering factors

Adaptable Process Model

Each of the adaptation criteria is assigned a grade that ranges between 1 and 5, where 1 represents a project in which a small subset of process tasks are required and overall methodological and documentation requirements are minimal, and 5 represents a project in which a complete set of process tasks should be applied and overall methodological and documentation requirements are substantial.

---

8 Emergency situations should be rare (they should not occur on more than 10 percent of all work conducted within the software engineering context). An emergency is not the same as a project with tight time constraints.

TABLE 7.1 COMPUTING THE TASK SET SELECTOR

Adaptation Criteria	Grade	Weight	Entry Point Multiplier					Product
			Conc.	NDev.	Enhan.	Maint.	Reeng.	
Size of project	_____	1.20	0	1	1	1	1	_____
Number of users	_____	1.10	0	1	1	1	1	_____
Business criticality	_____	1.10	0	1	1	1	1	_____
Longevity	_____	0.90	0	1	1	0	0	_____
Stability of requirements	_____	1.20	0	1	1	1	1	_____
Ease of communication	_____	0.90	1	1	1	1	1	_____
Maturity of technology	_____	0.90	1	1	0	0	1	_____
Performance constraints	_____	0.80	0	1	1	0	1	_____
Embedded/nonembedded	_____	1.20	1	1	1	0	1	_____
Project staffing	_____	1.00	1	1	1	1	1	_____
Interoperability	_____	1.10	0	1	1	1	1	_____
Reengineering factors	_____	1.20	0	0	0	0	1	_____
Task set selector (TSS)								_____

7.3.3 Computing a Task Set Selector Value

How do we choose the appropriate task set for our project?

To select the appropriate task set for a project, the following steps should be conducted:

1. Review each of the adaptation criteria in Section 7.3.2 and assign the appropriate grades (1 to 5) based on the characteristics of the project. These grades should be entered into Table 7.1.
2. Review the weighting factors assigned to each of the criteria. The value of a weighting factor ranges from 0.8 to 1.2 and provides an indication of the relative importance of a particular adaptation criterion to the types of software developed within the local environment. If modifications are required to better reflect local circumstances, they should be made.
3. Multiply the grade entered in Table 7.1 by the *weighting factor* and by the *entry point multiplier* for the type of project to be undertaken. The entry point multiplier takes on a value of 0 or 1 and indicates the relevance of the adaptation criterion to the project type. The result of the product

   grade x weighting factor x entry point multiplier

   is placed in the Product column of Table 7.1 for each adaptation criteria individually.
4. Compute the average of all entries in the Product column and place the result in the space marked *task set selector* (TSS). This value will be used to help select the task set that is most appropriate for the project.

TABLE 7.2 COMPUTING THE TASK SET SELECTOR—AN EXAMPLE

Adaptation Criteria	Grade	Weight	Entry Point Multiplier					Product
			Conc.	NDev.	Enhan.	Maint.	Reeng.	
Size of project	2	1.2	____	1	____	____	____	2.4
Number of users	3	1.1	____	1	____	____	____	3.3
Business criticality	4	1.1	____	1	____	____	____	4.4
Longevity	3	0.9	____	1	____	____	____	2.7
Stability of requirements	2	1.2	____	1	____	____	____	2.4
Ease of communication	2	0.9	____	1	____	____	____	1.8
Maturity of technology	2	0.9	____	1	____	____	____	1.8
Performance constraints	3	0.8	____	1	____	____	____	2.4
Embedded/nonembedded	3	1.2	____	1	____	____	____	3.6
Project staffing	2	1.0	____	1	____	____	____	2.0
Interoperability	4	1.1	____	1	____	____	____	4.4
Reengineering factors	0	1.2	____	0	____	____	____	0.0
Task set selector (TSS)								**2.8**

7.3.4 Interpreting the TSS Value and Selecting the Task Set

Once the task set selector is computed, the following guidelines can be used to select the appropriate task set for a project:

Task set selector value	Degree of rigor
TSS < 1.2	casual
1.0 < TSS < 3.0	structured
TSS > 2.4	strict

If the task set selector value is in an overlap area, it usually is OK to choose the less formal degree of rigor, unless project risk is high.

The overlap in TSS values from one recommended task set to another is purposeful and is intended to illustrate that sharp boundaries are impossible to define when making task set selections. In the final analysis, the task set selector value, past experience, and common sense must all be factored into the choice of the task set for a project.

Table 7.2 illustrates how TSS might be computed for a hypothetical project. The project manager selects the grades shown in the Grade column. The project type is *new application development.* Therefore, entry point multipliers are selected from the NDev column. The entry in the Product column is computed using

Grade x Weight x NewDev entry point multiplier

The value of TSS (computed as the average of all entries in the product column) is 2.8. Using the criteria discussed previously, the manager has the option of using either the structured or the strict task set. The final decision is made once all project factors have been considered.

7.4 SELECTING SOFTWARE ENGINEERING TASKS

An adaptable process model (APM) includes a variety of task sets and is available for your use.

In order to develop a project schedule, a task set must be distributed on the project time line. As we noted in Section 7.3, the task set will vary depending upon the project type and the degree of rigor. Each of the project types described in Section 7.3 may be approached using a process model that is linear sequential, iterative (e.g., the prototyping or incremental models), or evolutionary (e.g., the spiral model). In some cases, one project type flows smoothly into the next. For example, concept development projects that succeed often evolve into new application development projects. As a new application development project ends, an application enhancement project sometimes begins. This progression is both natural and predictable and will occur regardless of the process model that is adopted by an organization. Therefore, the major software engineering tasks described in the sections that follow are applicable to all process model flows. As an example, we consider the software engineering tasks for a concept development project.

Concept development projects are initiated when the potential for some new technology must be explored. There is no certainty that the technology will be applicable, but a customer (e.g., marketing) believes that potential benefit exists. Concept development projects are approached by applying the following major tasks:

Concept scoping determines the overall scope of the project.

Preliminary concept planning establishes the organization's ability to undertake the work implied by the project scope.

Technology risk assessment evaluates the risk associated with the technology to be implemented as part of project scope.

Proof of concept demonstrates the viability of a new technology in the software context.

Concept implementation implements the concept representation in a manner that can be reviewed by a customer and is used for "marketing" purposes when a concept must be sold to other customers or management.

Customer reaction to the concept solicits feedback on a new technology concept and targets specific customer applications.

A quick scan of these tasks should yield few surprises. In fact, the software engineering flow for concept development projects (and for all other types of projects as well) is little more than common sense.

The software team must understand what must be done (scoping); then the team (or manager) must determine whether anyone is available to do it (planning), consider the risks associated with the work (risk assessment), prove the technology in some way (proof of concept), and implement it in a prototypical manner so that the customer can evaluate it (concept implementation and customer evaluation). Finally, if the concept is viable, a production version (translation) must be produced.

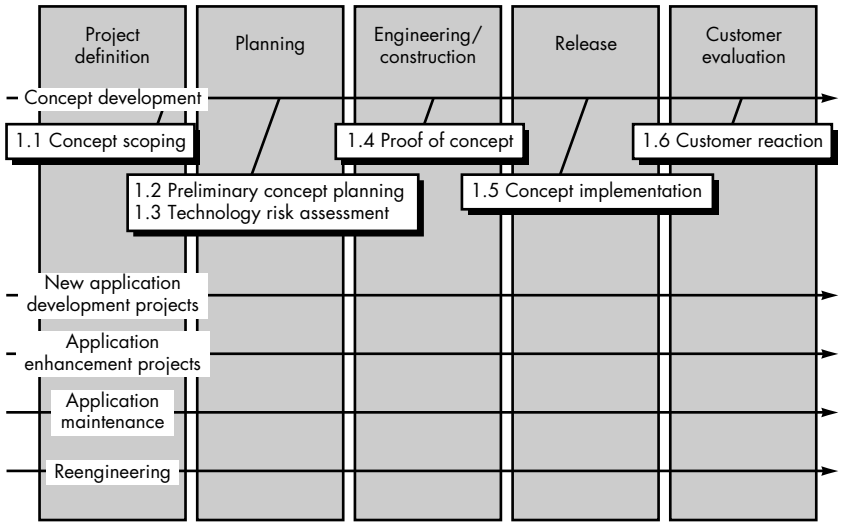


FIGURE 7.1 Concept development tasks in a linear sequential model

It is important to note that concept development framework activities are iterative in nature. That is, an actual concept development project might approach these activities in a number of planned increments, each designed to produce a deliverable that can be evaluated by the customer.

If a linear process model flow is chosen, each of these increments is defined in a repeating sequence as illustrated in Figure 7.1. During each sequence, umbrella activities (described in Chapter 2) are applied; quality is monitored; and at the end of each sequence, a deliverable is produced. With each iteration, the deliverable should converge toward the defined end product for the concept development stage. If an evolutionary model is chosen, the layout of tasks 1.1 through 1.6 would appear as shown in Figure 7.2. Major software engineering tasks for other project types can be defined and applied in a similar manner.

7.5 REFINEMENT OF MAJOR TASKS

The major tasks described in Section 7.4 may be used to define a macroscopic schedule for a project. However, the macroscopic schedule must be refined to create a detailed project schedule. Refinement begins by taking each major task and decomposing it into a set of subtasks (with related work products and milestones).

As an example of task decomposition, consider *concept scoping* for a development project, discussed in Section 7.4. Task refinement can be accomplished using an outline format, but in this book, a process design language approach is used to illustrate the flow of the concept scoping activity:

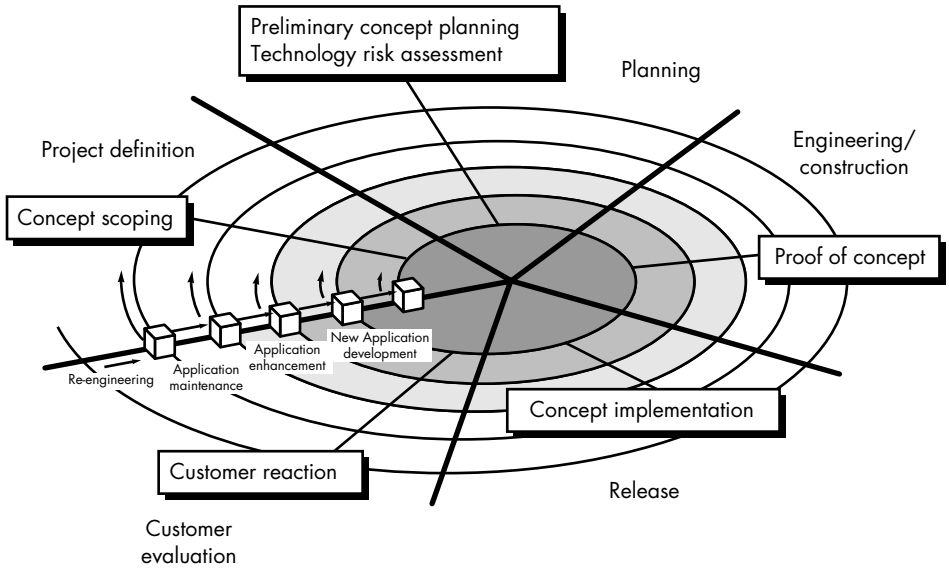


FIGURE 7.2
Concept development tasks using an evolutionary model

Task definition: Task I.1 Concept Scoping

I.1.1 Identify need, benefits and potential customers;
I.1.2 Define desired output/control and input events that drive the application;
Begin Task I.1.2
I.1.2.1 FTR: Review written description of need[9]
I.1.2.2 Derive a list of customer visible outputs/inputs
case of: mechanics
mechanics = quality function deployment
meet with customer to isolate major concept requirements;
interview end-users;
observe current approach to problem, current process;
review past requests and complaints;
mechanics = structured analysis
make list of major data objects;
define relationships between objects;
define object attributes;
mechanics = object view
make list of problem classes;
develop class hierarchy and class connections;
define attributes for classes;
endcase
I.1.2.3 FTR: Review outputs/inputs with customer and revise as required;
endtask Task I.1.2
I.1.3 Define the functionality/behavior for each major function;
Begin Task I.1.3

The adaptable process model (APM) contains a complete process design language description for all software engineering tasks.

9 FTR indicates that a formal technical review (Chapter 8) is to be conducted.

```
    I.1.3.1  FTR: Review output and input data objects derived in task I.1.2;
    I.1.3.2  Derive a model of functions/behaviors;
             case of: mechanics
             mechanics = quality function deployment
                    meet with customer to review major concept requirements;
                    interview end-users;
                    observe current approach to problem, current process;
                    develop a hierarchical outline of functions/behaviors;
             mechanics = structured analysis
                    derive a context level data flow diagram;
                    refine the data flow diagram to provide more detail;
                    write processing narratives for functions at lowest level of refinement;
             mechanics = object view
                    define operations/methods that are relevant for each class;
             endcase
    I.1.3.3  FTR: Review functions/behaviors with customer and revise as required;
    endtask Task I.1.3
I.1.4    Isolate those elements of the technology to be implemented in software;
I.1.5    Research availability of existing software;
I.1.6    Define technical feasibility;
I.1.7    Make quick estimate of size;
I.1.8    Create a Scope Definition;
endTask definition:  Task I.1
```

The tasks and subtasks noted in the process design language refinement form the basis for a detailed schedule for the concept scoping activity.

7.6 DEFINING A TASK NETWORK

The task network is a useful mechanism for depicting intertask dependencies and determining the critical path.

Individual tasks and subtasks have interdependencies based on their sequence. In addition, when more than one person is involved in a software engineering project, it is likely that development activities and tasks will be performed in parallel. When this occurs, concurrent tasks must be coordinated so that they will be complete when later tasks require their work product(s).

A *task network,* also called an *activity network,* is a graphic representation of the task flow for a project. It is sometimes used as the mechanism through which task sequence and dependencies are input to an automated project scheduling tool. In its simplest form (used when creating a macroscopic schedule), the task network depicts major software engineering tasks. Figure 7.3 shows a schematic task network for a concept development project.

The concurrent nature of software engineering activities leads to a number of important scheduling requirements. Because parallel tasks occur asynchronously, the planner must determine intertask dependencies to ensure continuous progress toward completion. In addition, the project manager should be aware of those tasks that lie on the critical path. That is, tasks that must be completed on schedule if the project

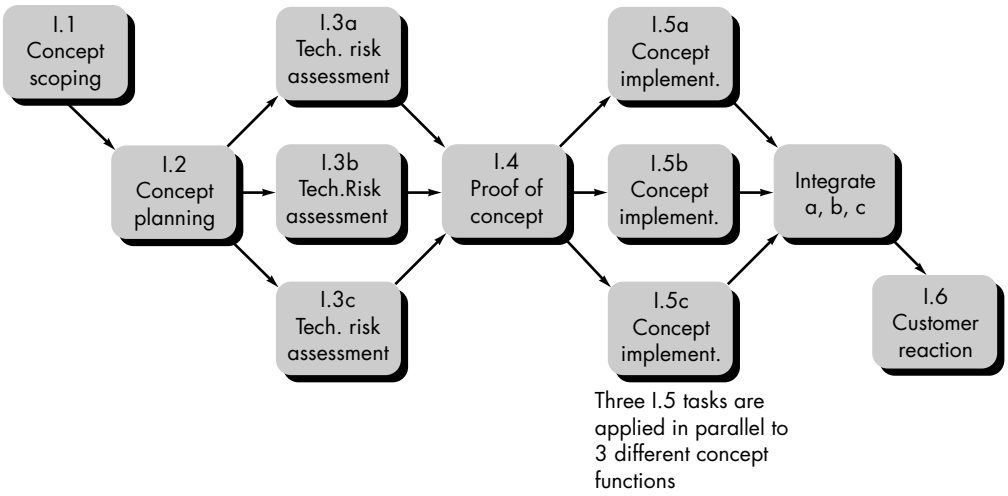


FIGURE 7.3 A task network for concept development

as a whole is to be completed on schedule. These issues are discussed in more detail later in this chapter.

It is important to note that the task network shown in Figure 7.3 is macroscopic. In a detailed task network (a precursor to a detailed schedule), each activity shown in Figure 7.3 would be expanded. For example, Task I.1 would be expanded to show all tasks detailed in the refinement of Tasks I.1 shown in Section 7.5.

7.7 SCHEDULING

For all but the simplest projects, scheduling should be done with the aid of a project scheduling tool.

Scheduling of a software project does not differ greatly from scheduling of any multitask engineering effort. Therefore, generalized project scheduling tools and techniques can be applied with little modification to software projects.

Program evaluation and review technique (PERT) and *critical path method* (CPM) [MOD83] are two project scheduling methods that can be applied to software development. Both techniques are driven by information already developed in earlier project planning activities:

- Estimates of effort
- A decomposition of the product function
- The selection of the appropriate process model and task set
- Decomposition of tasks

Interdependencies among tasks may be defined using a task network. Tasks, sometimes called the project *work breakdown structure* (WBS), are defined for the product as a whole or for individual functions.

Both PERT and CPM provide quantitative tools that allow the software planner to (1) determine the *critical path*—the chain of tasks that determines the duration of the

project; (2) establish "most likely" time estimates for individual tasks by applying statistical models; and (3) calculate "boundary times" that define a time "window" for a particular task.

Boundary time calculations can be very useful in software project scheduling. Slippage in the design of one function, for example, can retard further development of other functions. Riggs [RIG81] describes important boundary times that may be discerned from a PERT or CPM network: (1) the earliest time that a task can begin when all preceding tasks are completed in the shortest possible time, (2) the latest time for task initiation before the minimum project completion time is delayed, (3) the earliest finish—the sum of the earliest start and the task duration, (4) the latest finish—the latest start time added to task duration, and (5) the *total float*—the amount of surplus time or leeway allowed in scheduling tasks so that the network critical path is maintained on schedule. Boundary time calculations lead to a determination of critical path and provide the manager with a quantitative method for evaluating progress as tasks are completed.

CASE tools project/scheduling and planning

Both PERT and CPM have been implemented in a wide variety of automated tools that are available for the personal computer [THE93]. Such tools are easy to use and make the scheduling methods described previously available to every software project manager.

7.7.1 Timeline Charts

When creating a software project schedule, the planner begins with a set of tasks (the work breakdown structure). If automated tools are used, the work breakdown is input as a task network or task outline. Effort, duration, and start date are then input for each task. In addition, tasks may be assigned to specific individuals.

A timeline chart enables you to determine what tasks will be conducted at a given point in time.

As a consequence of this input, a *timeline chart,* also called a *Gantt chart,* is generated. A timeline chart can be developed for the entire project. Alternatively, separate charts can be developed for each project function or for each individual working on the project.

Figure 7.4 illustrates the format of a timeline chart. It depicts a part of a software project schedule that emphasizes the concept scoping task (Section 7.5) for a new word-processing (WP) software product. All project tasks (for concept scoping) are listed in the left-hand column. The horizontal bars indicate the duration of each task. When multiple bars occur at the same time on the calendar, task concurrency is implied. The diamonds indicate milestones.

Once the information necessary for the generation of a timeline chart has been input, the majority of software project scheduling tools produce *project tables*—a tabular listing of all project tasks, their planned and actual start- and end-dates, and a variety of related information (Figure 7.5). Used in conjunction with the timeline chart, project tables enable the project manager to track progress.

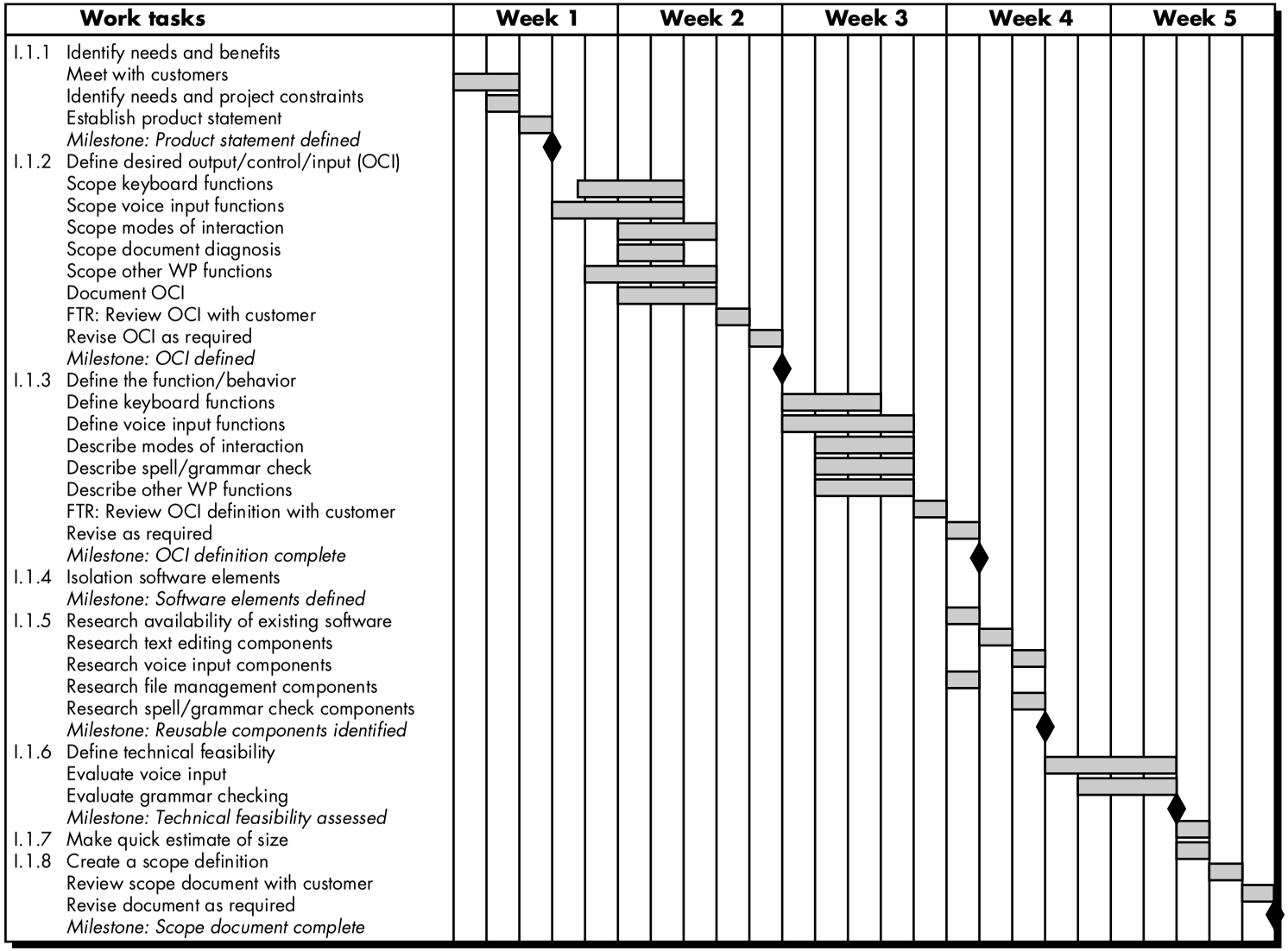


FIGURE 7.4 An example timeline chart

Work tasks	Planned start	Actual start	Planned complete	Actual complete	Assigned person	Effort allocated	Notes
I.1.1 Identify needs and benefits							Scoping will
Meet with customers	wk1, d1	wk1, d1	wk1, d2	wk1, d2	BLS	2 p-d	require more
Identify needs and project constraints	wk1, d2	wk1, d2	wk1, d2	wk1, d2	JPP	1 p-d	effort/time
Establish product statement	wk1, d3	wk1, d3	wk1, d3	wk1, d3	BLS/JPP	1 p-d	
Milestone: Product statement defined	wk1, d3	wk1, d3	wk1, d3	wk1, d3			
I.1.2 Define desired output/control/input (OCI)							
Scope keyboard functions	wk1, d4	wk1, d4	wk2, d2		BLS	1.5 p-d	
Scope voice input functions	wk1, d3	wk1, d3	wk2, d2		JPP	2 p-d	
Scope modes of interaction	wk2, d1		wk2, d3		MLL	1 p-d	
Scope document diagnostics	wk2, d1		wk2, d2		BLS	1.5 p-d	
Scope other WP functions	wk1, d4	wk1, d4	wk2, d3		JPP	2 p-d	
Document OCI	wk2, d1		wk2, d3		MLL	3 p-d	
FTR: Review OCI with customer	wk2, d3		wk2, d3		all	3 p-d	
Revise OCI as required	wk2, d4		wk2, d4		all	3 p-d	
Milestone: OCI defined	wk2, d5		wk2, d5				
I.1.3 Define the function/behavior							

FIGURE 7.5 An example project table

7.7.2 Tracking the Schedule

The project schedule provides a road map for a software project manager. If it has been properly developed, the project schedule defines the tasks and milestones that must be tracked and controlled as the project proceeds. Tracking can be accomplished in a number of different ways:

Quote:

"The basic rule of software status reporting can be summarized in a single phrase: 'No surprises!'."

Capers Jones

- Conducting periodic project status meetings in which each team member reports progress and problems.
- Evaluating the results of all reviews conducted throughout the software engineering process.
- Determining whether formal project milestones (the diamonds shown in Figure 7.4) have been accomplished by the scheduled date.
- Comparing actual start-date to planned start-date for each project task listed in the resource table (Figure 7.5).
- Meeting informally with practitioners to obtain their subjective assessment of progress to date and problems on the horizon.
- Using earned value analysis (Section 7.8) to assess progress quantitatively.

In reality, all of these tracking techniques are used by experienced project managers.

ADVICE

The best indicator of progress is the completion and successful review of a defined software work product.

Control is employed by a software project manager to administer project resources, cope with problems, and direct project staff. If things are going well (i.e., the project is on schedule and within budget, reviews indicate that real progress is being made and milestones are being reached), control is light. But when problems occur, the project manager must exercise control to reconcile them as quickly as possible. After a problem has been diagnosed,[10] additional resources may be focused on the problem area: staff may be redeployed or the project schedule can be redefined.

When faced with severe deadline pressure, experienced project managers sometimes use a project scheduling and control technique called *time-boxing* [ZAH95]. The time-boxing strategy recognizes that the complete product may not be deliverable by the predefined deadline. Therefore, an incremental software paradigm (Chapter 2) is chosen and a schedule is derived for each incremental delivery.

The tasks associated with each increment are then time-boxed. This means that the schedule for each task is adjusted by working backward from the delivery date for the increment. A "box" is put around each task. When a task hits the boundary of its time box (plus or minus 10 percent), work stops and the next task begins.

The initial reaction to the time-boxing approach is often negative: "If the work isn't finished, how can we proceed?" The answer lies in the way work is accomplished. By the time the time-box boundary is encountered, it is likely that 90 percent of the

10 It is important to note that schedule slippage is a symptom of some underlying problem. The role of the project manager is to diagnose the underlying problem and act to correct it.

task has been completed.[11] The remaining 10 percent, although important, can (1) be delayed until the next increment or (2) be completed later if required. Rather than becoming "stuck" on a task, the project proceeds toward the delivery date.

7.8 EARNED VALUE ANALYSIS

KEY POINT

Earned value provides a quantitative indication of progress.

In Section 7.7.2, we discussed a number of qualitative approaches to project tracking. Each provides the project manager with an indication of progress, but an assessment of the information provided is somewhat subjective. It is reasonable to ask whether there is a quantitative technique for assessing progress as the software team progresses through the work tasks allocated to the project schedule. In fact, a technique for performing quantitative analysis of progress does exist. It is called *earned value analysis* (EVA).

Humphrey [HUM95] discusses earned value in the following manner:

> The earned value system provides a common value scale for every [software project] task, regardless of the type of work being performed. The total hours to do the whole project are estimated, and every task is given an earned value based on its estimated percentage of the total.

Stated even more simply, earned value is a measure of progress. It enables us to assess the "percent of completeness" of a project using quantitative analysis rather than rely on a gut feeling. In fact, Fleming and Koppleman [FLE98] argue that earned value analysis "provides accurate and reliable readings of performance from as early as 15 percent into the project."

To determine the earned value, the following steps are performed:

? How do I compute earned value to assess progress?

1. The *budgeted cost of work scheduled* (BCWS) is determined for each work task represented in the schedule. During the estimation activity (Chapter 5), the work (in person-hours or person-days) of each software engineering task is planned. Hence, $BCWS_i$ is the effort planned for work task *i*. To determine progress at a given point along the project schedule, the value of BCWS is the sum of the $BCWS_i$ values for all work tasks that should have been completed by that point in time on the project schedule.
2. The BCWS values for all work tasks are summed to derive the budget at completion, BAC. Hence,

   $$BAC = \Sigma (BCWS_k) \text{ for all tasks } k$$

3. Next, the value for *budgeted cost of work performed* (BCWP) is computed. The value for BCWP is the sum of the BCWS values for all work tasks that have actually been completed by a point in time on the project schedule.

11 A cynic might recall the saying: "The first 90 percent of a system takes 90 percent of the time. The last 10 percent of the system takes 90 percent of the time."

Wilkens [WIL99] notes that "the distinction between the BCWS and the BCWP is that the former represents the budget of the activities that were planned to be completed and the latter represents the budget of the activities that actually were completed." Given values for BCWS, BAC, and BCWP, important progress indicators can be computed:

Schedule performance index, SPI = BCWP/BCWS

Schedule variance, SV = BCWP – BCWS

SPI is an indication of the efficiency with which the project is utilizing scheduled resources. An SPI value close to 1.0 indicates efficient execution of the project schedule. SV is simply an absolute indication of variance from the planned schedule.

A wide array of earned value analysis resources (comprehensive bibliography, papers, hotlinks) can be found at **www.acq.osd.mil/pm/**

Percent scheduled for completion = BCWS/BAC

provides an indication of the percentage of work that should have been completed by time *t.*

Percent complete = BCWP/BAC

provides a quantitative indication of the percent of completeness of the project at a given point in time, *t.*

It is also possible to compute the *actual cost of work performed,* ACWP. The value for ACWP is the sum of the effort actually expended on work tasks that have been completed by a point in time on the project schedule. It is then possible to compute

Cost performance index, CPI = BCWP/ACWP

Cost variance, CV = BCWP – ACWP

A CPI value close to 1.0 provides a strong indication that the project is within its defined budget. CV is an absolute indication of cost savings (against planned costs) or shortfall at a particular stage of a project.

Like over-the-horizon radar, earned value analysis illuminates scheduling difficulties before they might otherwise be apparent. This enables the software project manager to take corrective action before a project crisis develops.

7.9 ERROR TRACKING

Error tracking allows you to compare current work with past efforts and provides a quantitative indication of the quality of the work being conducted.

Throughout the software process, a project team creates work products (e.g., requirements specifications or prototype, design documents, source code). But the team also creates (and hopefully corrects) errors associated with each work product. If error-related measures and resultant metrics are collected over many software projects, a project manager can use these data as a baseline for comparison against error data collected in real time. Error tracking can be used as one means for assessing the status of a current project.

In Chapter 4, the concept of defect removal efficiency was discussed. To review briefly, the software team performs formal technical reviews (and, later, testing) to find and correct errors, *E,* in work products produced during software engineering

tasks. Any errors that are not uncovered (but found in later tasks) are considered to be defects, *D.* Defect removal efficiency (Chapter 4) has been defined as

$$\text{DRE} = E/(E + D)$$

DRE is a process metric that provides a strong indication of the effectiveness of quality assurance activities, but DRE and the error and defect counts associated with it can also be used to assist a project manager in determining the progress that is being made as a software project moves through its scheduled work tasks.

Let us assume that a software organization has collected error and defect data over the past 24 months and has developed averages for the following metrics:

- Errors per requirements specification page, E_{req}
- Errors per component—design level, E_{design}
- Errors per component—code level, E_{code}
- DRE—requirements analysis
- DRE—architectural design
- DRE—component level design
- DRE—coding

As the project progresses through each software engineering step, the software team records and reports the number of errors found during requirements, design, and code reviews. The project manager calculates current values for E_{req}, E_{design}, and E_{code}. These are then compared to averages for past projects. If current results vary by more than 20% from the average, there may be cause for concern and there is certainly cause for investigation.

For example, if $E_{\text{req}} = 2.1$ for project X, yet the organizational average is 3.6, one of two scenarios is possible: (1) the software team has done an outstanding job of developing the requirements specification or (2) the team has been lax in its review approach. If the second scenario appears likely, the project manager should take immediate steps to build additional design time[12] into the schedule to accommodate the requirements defects that have likely been propagated into the design activity.

The more quantitative your approach to project tracking and control, the more likely you'll be able to foresee potential problems and respond to them proactively. Use earned value and tracking metrics.

These error tracking metrics can also be used to better target review and/or testing resources. For example, if a system is composed of 120 components, but 32 of these component exhibit E_{design} values that have substantial variance from the average, the project manager might elect to dedicate code review resources to the 32 components and allow others to pass into testing with no code review. Although all components should undergo code review in an ideal setting, a selective approach (reviewing only those modules that have suspect quality based on the E_{design} value) might be an effective means for recouping lost time and/or saving costs for a project that has gone over budget.

12 In reality, the extra time will be spent reworking requirements defects, but the work will occur when the design is underway.

7.10 THE PROJECT PLAN

Software Project Plan

Each step in the software engineering process should produce a deliverable that can be reviewed and that can act as a foundation for the steps that follow. The *Software Project Plan* is produced at the culmination of the planning tasks. It provides baseline cost and scheduling information that will be used throughout the software process.

The *Software Project Plan* is a relatively brief document that is addressed to a diverse audience. It must (1) communicate scope and resources to software management, technical staff, and the customer; (2) define risks and suggest risk aversion techniques; (3) define cost and schedule for management review; (4) provide an overall approach to software development for all people associated with the project; and (5) outline how quality will be ensured and change will be managed.

A presentation of cost and schedule will vary with the audience addressed. If the plan is used only as an internal document, the results of each estimation technique can be presented. When the plan is disseminated outside the organization, a reconciled cost breakdown (combining the results of all estimation techniques) is provided. Similarly, the degree of detail contained within the schedule section may vary with the audience and formality of the plan.

It is important to note that the *Software Project Plan* is *not* a static document. That is, the project team revisits the plan repeatedly—updating risks, estimates, schedules and related information—as the project proceeds and more is learned.

7.11 SUMMARY

Scheduling is the culmination of a planning activity that is a primary component of software project management. When combined with estimation methods and risk analysis, scheduling establishes a road map for the project manager.

Scheduling begins with process decomposition. The characteristics of the project are used to adapt an appropriate task set for the work to be done. A task network depicts each engineering task, its dependency on other tasks, and its projected duration. The task network is used to compute the critical path, a timeline chart and a variety of project information. Using the schedule as a guide, the project manager can track and control each step in the software process.

REFERENCES

[BRO95] Brooks, M., *The Mythical Man-Month,* Anniversary Edition, Addison-Wesley, 1995.

[FLE98] Fleming, Q.W. and J.M. Koppelman, "Earned Value Project Management," *Crosstalk,* vol. 11, no. 7, July 1998, p. 19.

[HUM95] Humphrey, W., *A Discipline for Software Engineering,* Addison-Wesley, 1995.

[PAG85] Page-Jones, M., *Practical Project Management,* Dorset House, 1985, pp. 90–91.

[PRE99] Pressman, R.S., *Adaptable Process Model,* R.S. Pressman & Associates, 1999.
[PUT92] Putnam, L. and W. Myers, *Measures for Excellence,* Yourdon Press, 1992.
[RIG81] Riggs, J., *Production Systems Planning, Analysis and Control,* 3rd ed., Wiley, 1981.
[THE93] The', L., "Project Management Software That's IS Friendly," *Datamation,* October 1, 1993, pp. 55–58.
[WIL99] Wilkens, T.T., "Earned Value, Clear and Simple," Primavera Systems, April 1, 1999, p. 2.
[ZAH95] Zahniser, R., "Time-Boxing for Top Team Performance," *Software Development,* March 1995, pp. 34–38.

PROBLEMS AND POINTS TO PONDER

7.1. "Unreasonable" deadlines are a fact of life in the software business. How should you proceed if you're faced with one?

7.2. What is the difference between a macroscopic schedule and a detailed schedule. Is it possible to manage a project if only a macroscopic schedule is developed? Why?

7.3. Is there ever a case where a software project milestone is not tied to a review? If so, provide one or more examples.

7.4. In Section 7.2.1, we present an example of the "communication overhead" that can occur when multiple people work on a software project. Develop a counterexample that illustrates how engineers who are well-versed in good software engineering practices and use formal technical reviews can increase the production rate of a team (when compared to the sum of individual production rates). Hint: You can assume that reviews reduce rework and that rework can account for 20–40 percent of a person's time.

7.5. Although adding people to a late software project can make it later, there are circumstances in which this is not true. Describe them.

7.6. The relationship between people and time is highly nonlinear. Using Putnam's software equation (described in Section 7.2.2), develop a table that relates number of people to project duration for a software project requiring 50,000 LOC and 15 person-years of effort (the productivity parameter is 5000 and B = 0.37). Assume that the software must be delivered in 24 months plus or minus 12 months.

7.7. Assume that you have been contracted by a university to develop an on-line course registration system (OLCRS). First, act as the customer (if you're a student, that should be easy!) and specify the characteristics of a good system. (Alternatively, your instructor will provide you with a set of preliminary requirements for the system.) Using the estimation methods discussed in Chapter 5, develop an effort and duration estimate for OLCRS. Suggest how you would:

a. Define parallel work activities during the OLCRS project.
b. Distribute effort throughout the project.
c. Establish milestones for the project.

7.8. Using Section 7.3 as a guide compute the TSS for OLCRS. Be sure to show all of your work. Select a project type and an appropriate task set for the project.

7.9. Define a task network for OLCRS, or alternatively, for another software project that interests you. Be sure to show tasks and milestones and to attach effort and duration estimates to each task. If possible, use an automated scheduling tool to perform this work.

7.10. If an automated scheduling tool is available, determine the critical path for the network defined in problem 7.7.

7.11. Using a scheduling tool (if available) or paper and pencil (if necessary), develop a timeline chart for the OLCRS project.

7.12. Refine the task called "technology risk assessment" in Section 7.4 in much the same way as concept scoping was refined in Section 7.5.

7.13. Assume you are a software project manager and that you've been asked to compute earned value statistics for a small software project. The project has 56 planned work tasks that are estimated to require 582 person-days to complete. At the time that you've been asked to do the earned value analysis, 12 tasks have been completed. However the project schedule indicates that 15 tasks should have been completed. The following scheduling data (in person-days) are available:

Task	Planned effort	Actual effort
1	12.0	12.5
2	15.0	11.0
3	13.0	17.0
4	8.0	9.5
5	9.5	9.0
6	18.0	19.0
7	10.0	10.0
8	4.0	4.5
9	12.0	10.0
10	6.0	6.5
11	5.0	4.0
12	14.0	14.5
13	16.0	—
14	6.0	—
15	8.0	—

Compute the SPI, schedule variance, percent scheduled for completion, percent complete, CPI, and cost variance for the project.

7.14. Is it possible to use DRE as a metric for error tracking throughout a software project? Discuss the pros and cons of using DRE for this purpose.

FURTHER READINGS AND INFORMATION SOURCES

McConnell (*Rapid Development,* Microsoft Press, 1996) presents an excellent discussion of the issues that lead to overly optimistic software project scheduling and what you can do about it. O'Connell (*How to Run Successful Projects II: The Silver Bullet,* Prentice-Hall, 1997) presents a step-by-step approach to project management that will help you to develop a realistic schedule for your projects.

Project scheduling issues are covered in most books on software project management. McConnell (*Software Project Survival Guide,* Microsoft Press, 1998), Hoffman and Beaumont (*Application Development: Managing a Project's Life Cycle,* Midrange Computing, 1997), Wysoki and his colleagues (*Effective Project Management,* Wiley, 1995), and Whitten (*Managing Software Development Projects,* 2nd ed., Wiley, 1995) consider the topic in detail. Boddie (*Crunch Mode,* Prentice-Hall, 1987) has written a book for all managers who "have 90 days to do a six month project."

Worthwhile information on project scheduling can also be obtained in general purpose project management books. Among the many offerings available are

Kerzner, H., *Project Management: A Systems Approach to Planning, Scheduling, and Controlling,* Wiley, 1998.

Lewis, J.P., *Mastering Project Management: Applying Advanced Concepts of Systems Thinking, Control and Evaluation,* McGraw-Hill, 1998.

Fleming and Koppelman (*Earned Value Project Management,* Project Management Institute Publications, 1996) discuss the use of earned value techniques for project tracking and control in considerable detail.

A wide variety of information sources on project scheduling and management is available on the Internet. An up-to-date list of World Wide Web references that are relevant to scheduling can be found at the SEPA Web site:

http://www.mhhe.com/engcs/compsci/pressman/resources/project-sched.mhtml

CHAPTER

8 SOFTWARE QUALITY ASSURANCE

KEY CONCEPTS

defect amplification . . 204
formal technical reviews. 205
ISO 9000 216
poka yoke. 214
quality. 195
quality costs. . . . 196
software safety. 213
SQA 199
SQA activities . . 201
SQA plan 218
statistical SQA. . 209
variation. 194

The software engineering approach described in this book works toward a single goal: to produce high-quality software. Yet many readers will be challenged by the question: "What is software quality?"

Philip Crosby [CRO79], in his landmark book on quality, provides a wry answer to this question:

> The problem of quality management is not what people don't know about it. The problem is what they think they do know . . .
>
> In this regard, quality has much in common with sex. Everybody is for it. (Under certain conditions, of course.) Everyone feels they understand it. (Even though they wouldn't want to explain it.) Everyone thinks execution is only a matter of following natural inclinations. (After all, we do get along somehow.) And, of course, most people feel that problems in these areas are caused by other people. (If only they would take the time to do things right.)

Some software developers continue to believe that software quality is something you begin to worry about after code has been generated. Nothing could be further from the truth! *Software quality assurance* (SQA) is an umbrella activity (Chapter 2) that is applied throughout the software process.

QUICK LOOK

What is it? It's not enough to talk the talk by saying that software quality is important, you have to (1) explicitly define what is meant when you say "software quality," (2) create a set of activities that will help ensure that every software engineering work product exhibits high quality, (3) perform quality assurance activities on every software project, (4) use metrics to develop strategies for improving your software process and, as a consequence, the quality of the end product.

Who does it? Everyone involved in the software engineering process is responsible for quality.

Why is it important? You can do it right, or you can do it over again. If a software team stresses quality in all software engineering activities, it reduces the amount of rework that it must do. That results in lower costs, and more importantly, improved time-to-market.

What are the steps? Before software quality assurance activities can be initiated, it is important to define 'software quality' at a number of different levels of abstraction. Once you understand what quality is, a software team must identify a set of SQA activities that will filter errors out of work products before they are passed on.

What is the work product? A Software Quality Assurance Plan is created to define a software team's SQA strategy. During analysis, design, and code generation, the primary SQA work product is the formal technical review summary report. During ▶

QUICK LOOK

testing, test plans and procedures are produced. Other work products associated with process improvement may also be generated.

How do I ensure that I've done it right? Find errors before they become defects! That is, work to improve your defect removal efficiency (Chapters 4 and 7), thereby reducing the amount of rework that your software team has to perform.

SQA encompasses (1) a quality management approach, (2) effective software engineering technology (methods and tools), (3) formal technical reviews that are applied throughout the software process, (4) a multitiered testing strategy, (5) control of software documentation and the changes made to it, (6) a procedure to ensure compliance with software development standards (when applicable), and (7) measurement and reporting mechanisms.

In this chapter, we focus on the management issues and the process-specific activities that enable a software organization to ensure that it does "the right things at the right time in the right way."

8.1 QUALITY CONCEPTS[1]

It has been said that no two snowflakes are alike. Certainly when we watch snow falling it is hard to imagine that snowflakes differ at all, let alone that each flake possesses a unique structure. In order to observe differences between snowflakes, we must examine the specimens closely, perhaps using a magnifying glass. In fact, the closer we look, the more differences we are able to observe.

Quote:

"People forget how fast you did a job — but they always remember how well you did it."
Howard Newton.

This phenomenon, *variation between samples,* applies to all products of human as well as natural creation. For example, if two "identical" circuit boards are examined closely enough, we may observe that the copper pathways on the boards differ slightly in geometry, placement, and thickness. In addition, the location and diameter of the holes drilled in the boards varies as well.

All engineered and manufactured parts exhibit variation. The variation between samples may not be obvious without the aid of precise equipment to measure the geometry, electrical characteristics, or other attributes of the parts. However, with sufficiently sensitive instruments, we will likely come to the conclusion that no two samples of any item are exactly alike.

Variation control is the heart of quality control. A manufacturer wants to minimize the variation among the products that are produced, even when doing something relatively simple like duplicating diskettes. Surely, this cannot be a problem—duplicat-

1 This section, written by Michael Stovsky, has been adapted from "Fundamentals of ISO 9000," a workbook developed for *Essential Software Engineering,* a video curriculum developed by R. S. Pressman & Associates, Inc. Reprinted with permission.

ing diskettes is a trivial manufacturing operation, and we can guarantee that exact duplicates of the software are always created.

Or can we? We need to ensure the tracks are placed on the diskettes within a specified tolerance so that the overwhelming majority of disk drives can read the diskettes. In addition, we need to ensure the magnetic flux for distinguishing a zero from a one is sufficient for read/write heads to detect. The disk duplication machines can, and do, wear and go out of tolerance. So even a "simple" process such as disk duplication may encounter problems due to variation between samples.

KEY POINT

Controlling variation is the key to a high-quality product. In the software context, we strive to control the variation in the process we apply, the resources we expend, and the quality attributes of the end product.

But how does this apply to software work? How might a software development organization need to control variation? From one project to another, we want to minimize the difference between the predicted resources needed to complete a project and the actual resources used, including staff, equipment, and calendar time. In general, we would like to make sure our testing program covers a known percentage of the software, from one release to another. Not only do we want to minimize the number of defects that are released to the field, we'd like to ensure that the variance in the number of bugs is also minimized from one release to another. (Our customers will likely be upset if the third release of a product has ten times as many defects as the previous release.) We would like to minimize the differences in speed and accuracy of our hotline support responses to customer problems. The list goes on and on.

8.1.1 Quality

The *American Heritage Dictionary* defines *quality* as "a characteristic or attribute of something." As an attribute of an item, quality refers to measurable characteristics—things we are able to compare to known standards such as length, color, electrical properties, and malleability. However, software, largely an intellectual entity, is more challenging to characterize than physical objects.

Nevertheless, measures of a program's characteristics do exist. These properties include cyclomatic complexity, cohesion, number of function points, lines of code, and many others, discussed in Chapters 19 and 24. When we examine an item based on its measurable characteristics, two kinds of quality may be encountered: quality of design and quality of conformance.

Quote:

"It takes less time to do a thing right than explain why you did it wrong."
Henry Wadsworth Longfellow

Quality of design refers to the characteristics that designers specify for an item. The grade of materials, tolerances, and performance specifications all contribute to the quality of design. As higher-grade materials are used, tighter tolerances and greater levels of performance are specified, the design quality of a product increases, if the product is manufactured according to specifications.

Quality of conformance is the degree to which the design specifications are followed during manufacturing. Again, the greater the degree of conformance, the higher is the level of quality of conformance.

In software development, quality of design encompasses requirements, specifications, and the design of the system. Quality of conformance is an issue focused

primarily on implementation. If the implementation follows the design and the resulting system meets its requirements and performance goals, conformance quality is high.

But are quality of design and quality of conformance the only issues that software engineers must consider? Robert Glass [GLA98] argues that a more "intuitive" relationship is in order:

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

At the bottom line, Glass contends that quality is important, but if the user isn't satisfied, nothing else really matters. DeMarco [DEM99] reinforces this view when he states: "A product's quality is a function of how much it changes the world for the better." This view of quality contends that if a software product provides substantial benefit to its end-users, they may be willing to tolerate occasional reliability or performance problems.

8.1.2 Quality Control

What is software quality control?

Variation control may be equated to quality control. But how do we achieve quality control? *Quality control* involves the series of inspections, reviews, and tests used throughout the software process to ensure each work product meets the requirements placed upon it. Quality control includes a feedback loop to the process that created the work product. The combination of measurement and feedback allows us to tune the process when the work products created fail to meet their specifications. This approach views quality control as part of the manufacturing process.

Quality control activities may be fully automated, entirely manual, or a combination of automated tools and human interaction. A key concept of quality control is that all work products have defined, measurable specifications to which we may compare the output of each process. The feedback loop is essential to minimize the defects produced.

8.1.3 Quality Assurance

WebRef

A wide variety of software quality resources can be found at **www.qualitytree.com/links/links.htm**

Quality assurance consists of the auditing and reporting functions of management. The goal of quality assurance is to provide management with the data necessary to be informed about product quality, thereby gaining insight and confidence that product quality is meeting its goals. Of course, if the data provided through quality assurance identify problems, it is management's responsibility to address the problems and apply the necessary resources to resolve quality issues.

8.1.4 Cost of Quality

The *cost of quality* includes all costs incurred in the pursuit of quality or in performing quality-related activities. Cost of quality studies are conducted to provide a base-

line for the current cost of quality, identify opportunities for reducing the cost of quality, and provide a normalized basis of comparison. The basis of normalization is almost always dollars. Once we have normalized quality costs on a dollar basis, we have the necessary data to evaluate where the opportunities lie to improve our processes. Furthermore, we can evaluate the effect of changes in dollar-based terms.

What are the components of the cost of quality?

Quality costs may be divided into costs associated with prevention, appraisal, and failure. *Prevention costs* include

- quality planning
- formal technical reviews
- test equipment
- training

Appraisal costs include activities to gain insight into product condition the "first time through" each process. Examples of appraisal costs include

- in-process and interprocess inspection
- equipment calibration and maintenance
- testing

Don't be afraid to incur significant prevention costs. Rest assured that your investment will provide an excellent return.

Failure costs are those that would disappear if no defects appeared before shipping a product to customers. Failure costs may be subdivided into internal failure costs and external failure costs. *Internal failure costs* are incurred when we detect a defect in our product prior to shipment. Internal failure costs include

- rework
- repair
- failure mode analysis

External failure costs are associated with defects found after the product has been shipped to the customer. Examples of external failure costs are

- complaint resolution
- product return and replacement
- help line support
- warranty work

As expected, the relative costs to find and repair a defect increase dramatically as we go from prevention to detection to internal failure to external failure costs. Figure 8.1, based on data collected by Boehm [BOE81] and others, illustrates this phenomenon.

Anecdotal data reported by Kaplan, Clark, and Tang [KAP95] reinforces earlier cost statistics and is based on work at IBM's Rochester development facility:

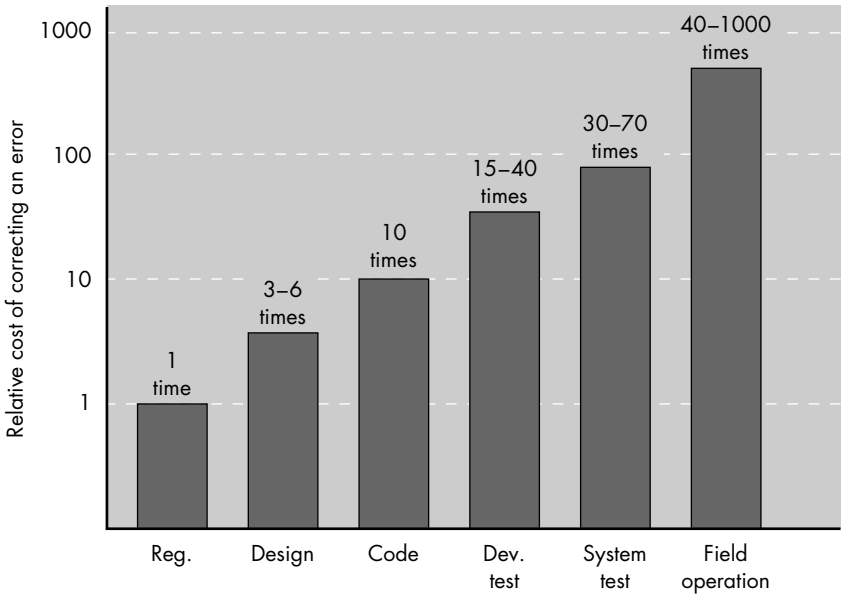


FIGURE 8.1 Relative cost of correcting an error

Testing is necessary, but it's also a very expensive way to find errors. Spend time finding errors early in the process and you may be able to significantly reduce testing and debugging costs.

A total of 7053 hours was spent inspecting 200,000 lines of code with the result that 3112 potential defects were prevented. Assuming a programmer cost of $40.00 per hour, the total cost of preventing 3112 defects was $282,120, or roughly $91.00 per defect.

Compare these numbers to the cost of defect removal once the product has been shipped to the customer. Suppose that there had been no inspections, but that programmers had been extra careful and only one defect per 1000 lines of code [significantly better than industry average] escaped into the shipped product. That would mean that 200 defects would still have to be fixed in the field. At an estimated cost of $25,000 per field fix, the cost would be $5 million, or approximately 18 times more expensive than the total cost of the defect prevention effort.

It is true that IBM produces software that is used by hundreds of thousands of customers and that their costs for field fixes may be higher than those for software organizations that build custom systems. This in no way negates the results just noted. Even if the average software organization has field fix costs that are 25 percent of IBM's (most have no idea what their costs are!), the cost savings associated with quality control and assurance activities are compelling.

8.2 THE QUALITY MOVEMENT

Today, senior managers at companies throughout the industrialized world recognize that high product quality translates to cost savings and an improved bottom line. However, this was not always the case. The quality movement began in the 1940s with the seminal work of W. Edwards Deming [DEM86] and had its first true test in Japan. Using Deming's ideas as a cornerstone, the Japanese developed a systematic

TQM can be applied to computer software. The TQM approach stresses continuous process improvement.

approach to the elimination of the root causes of product defects. Throughout the 1970s and 1980s, their work migrated to the western world and was given names such as "total quality management" (TQM).[2] Although terminology differs across different companies and authors, a basic four step progression is normally encountered and forms the foundation of any good TQM program.

The first step, called *kaizen,* refers to a system of continuous process improvement. The goal of *kaizen* is to develop a process (in this case, the software process) that is visible, repeatable, and measurable.

The second step, invoked only after *kaizen* has been achieved, is called *atarimae hinshitsu.* This step examines intangibles that affect the process and works to optimize their impact on the process. For example, the software process may be affected by high staff turnover, which itself is caused by constant reorganization within a company. Maybe a stable organizational structure could do much to improve the quality of software. *Atarimae hinshitsu* would lead management to suggest changes in the way reorganization occurs.

A wide variety of resources for continuous process improvement and TQM can be found at **deming.eng.clemson.edu/**

While the first two steps focus on the process, the next step, called *kansei* (translated as "the five senses"), concentrates on the user of the product (in this case, software). In essence, by examining the way the user applies the product *kansei* leads to improvement in the product itself and, potentially, to the process that created it.

Finally, a step called *miryokuteki hinshitsu* broadens management concern beyond the immediate product. This is a business-oriented step that looks for opportunity in related areas identified by observing the use of the product in the marketplace. In the software world, *miryokuteki hinshitsu* might be viewed as an attempt to uncover new and profitable products or applications that are an outgrowth from an existing computer-based system.

For most companies *kaizen* should be of immediate concern. Until a mature software process (Chapter 2) has been achieved, there is little point in moving to the next steps.

8.3 SOFTWARE QUALITY ASSURANCE

Even the most jaded software developers will agree that high-quality software is an important goal. But how do we define quality? A wag once said, "Every program does something right, it just may not be the thing that we want it to do."

Many definitions of software quality have been proposed in the literature. For our purposes, *software quality* is defined as

How do we define software quality?

Conformance to explicitly stated functional and performance requirements, explicitly documented development standards, and implicit characteristics that are expected of all professionally developed software.

---

2 See [ART92] for a comprehensive discussion of TQM and its use in a software context and [KAP95] for a discussion of the use of the Baldrige Award criteria in the software world.

There is little question that this definition could be modified or extended. In fact, a definitive definition of software quality could be debated endlessly. For the purposes of this book, the definition serves to emphasize three important points:

1. Software requirements are the foundation from which quality is measured. Lack of conformance to requirements is lack of quality.
2. Specified standards define a set of development criteria that guide the manner in which software is engineered. If the criteria are not followed, lack of quality will almost surely result.
3. A set of implicit requirements often goes unmentioned (e.g., the desire for ease of use and good maintainability). If software conforms to its explicit requirements but fails to meet implicit requirements, software quality is suspect.

8.3.1 Background Issues

Quote:

"We made too many wrong mistakes."

Yogi Berra

Quality assurance is an essential activity for any business that produces products to be used by others. Prior to the twentieth century, quality assurance was the sole responsibility of the craftsperson who built a product. The first formal quality assurance and control function was introduced at Bell Labs in 1916 and spread rapidly throughout the manufacturing world. During the 1940s, more formal approaches to quality control were suggested. These relied on measurement and continuous process improvement as key elements of quality management.

Today, every company has mechanisms to ensure quality in its products. In fact, explicit statements of a company's concern for quality have become a marketing ploy during the past few decades.

WebRef

An in-depth tutorial and wide-ranging resources for quality management can be found at **www.management.gov.**

The history of quality assurance in software development parallels the history of quality in hardware manufacturing. During the early days of computing (1950s and 1960s), quality was the sole responsibility of the programmer. Standards for quality assurance for software were introduced in military contract software development during the 1970s and have spread rapidly into software development in the commercial world [IEE94]. Extending the definition presented earlier, software quality assurance is a "planned and systematic pattern of actions" [SCH98] that are required to ensure high quality in software. The scope of quality assurance responsibility might best be characterized by paraphrasing a once-popular automobile commercial: "Quality Is Job #1." The implication for software is that many different constituencies have software quality assurance responsibility—software engineers, project managers, customers, salespeople, and the individuals who serve within an SQA group.

The SQA group serves as the customer's in-house representative. That is, the people who perform SQA must look at the software from the customer's point of view. Does the software adequately meet the quality factors noted in Chapter 19? Has soft-

ware development been conducted according to pre-established standards? Have technical disciplines properly performed their roles as part of the SQA activity? The SQA group attempts to answer these and other questions to ensure that software quality is maintained.

8.3.2 SQA Activities

Software quality assurance is composed of a variety of tasks associated with two different constituencies—the software engineers who do technical work and an SQA group that has responsibility for quality assurance planning, oversight, record keeping, analysis, and reporting.

Software engineers address quality (and perform quality assurance and quality control activities) by applying solid technical methods and measures, conducting formal technical reviews, and performing well-planned software testing. Only reviews are discussed in this chapter. Technology topics are discussed in Parts Three through Five of this book.

The charter of the SQA group is to assist the software team in achieving a high-quality end product. The Software Engineering Institute [PAU93] recommends a set of SQA activities that address quality assurance planning, oversight, record keeping, analysis, and reporting. These activities are performed (or facilitated) by an independent SQA group that:

What is the role of an SQA group?

Prepares an SQA plan for a project. The plan is developed during project planning and is reviewed by all interested parties. Quality assurance activities performed by the software engineering team and the SQA group are governed by the plan. The plan identifies

- evaluations to be performed
- audits and reviews to be performed
- standards that are applicable to the project
- procedures for error reporting and tracking
- documents to be produced by the SQA group
- amount of feedback provided to the software project team

Participates in the development of the project's software process description. The software team selects a process for the work to be performed. The SQA group reviews the process description for compliance with organizational policy, internal software standards, externally imposed standards (e.g., ISO-9001), and other parts of the software project plan.

Reviews software engineering activities to verify compliance with the defined software process. The SQA group identifies, documents, and tracks deviations from the process and verifies that corrections have been made.

Audits designated software work products to verify compliance with those defined as part of the software process. The SQA group reviews selected work products; identifies, documents, and tracks deviations; verifies that corrections have been made; and periodically reports the results of its work to the project manager.

Ensures that deviations in software work and work products are documented and handled according to a documented procedure. Deviations may be encountered in the project plan, process description, applicable standards, or technical work products.

Records any noncompliance and reports to senior management. Noncompliance items are tracked until they are resolved.

In addition to these activities, the SQA group coordinates the control and management of change (Chapter 9) and helps to collect and analyze software metrics.

8.4 SOFTWARE REVIEWS

Like water filters, FTRs tend to retard the "flow" of software engineering activities. Too few and the flow is "dirty." Too many and the flow slows to a trickle. Use metrics to determine which reviews work and which may not be effective. Take the ineffective ones out of the flow.

Software reviews are a "filter" for the software engineering process. That is, reviews are applied at various points during software development and serve to uncover errors and defects that can then be removed. Software reviews "purify" the software engineering activities that we have called *analysis, design,* and *coding.* Freedman and Weinberg [FRE90] discuss the need for reviews this way:

> Technical work needs reviewing for the same reason that pencils need erasers: *To err is human.* The second reason we need technical reviews is that although people are good at catching some of their own errors, large classes of errors escape the originator more easily than they escape anyone else. The review process is, therefore, the answer to the prayer of Robert Burns:
>
> *O wad some power the giftie give us*
> *to see ourselves as other see us*
>
> A review—any review—is a way of using the diversity of a group of people to:
>
> 1. Point out needed improvements in the product of a single person or team;
>
> 2. Confirm those parts of a product in which improvement is either not desired or not needed;
>
> 3. Achieve technical work of more uniform, or at least more predictable, quality than can be achieved without reviews, in order to make technical work more manageable.

Many different types of reviews can be conducted as part of software engineering. Each has its place. An informal meeting around the coffee machine is a form of review, if technical problems are discussed. A formal presentation of software design to an audience of customers, management, and technical staff is also a form of

review. In this book, however, we focus on the *formal technical review,* sometimes called a *walkthrough* or an *inspection.* A formal technical review is the most effective filter from a quality assurance standpoint. Conducted by software engineers (and others) for software engineers, the FTR is an effective means for improving software quality.

8.4.1 Cost Impact of Software Defects

The *IEEE Standard Dictionary of Electrical and Electronics Terms* (IEEE Standard 100-1992) defines a *defect* as "a product anomaly." The definition for *fault* in the hardware context can be found in IEEE Standard 610.12-1990:

> (a) A defect in a hardware device or component; for example, a short circuit or broken wire. (b) An incorrect step, process, or data definition in a computer program. Note: This definition is used primarily by the fault tolerance discipline. In common usage, the terms "error" and "bug" are used to express this meaning. See also: data-sensitive fault; program-sensitive fault; equivalent faults; fault masking; intermittent fault.

Within the context of the software process, the terms *defect* and *fault* are synonymous. Both imply a quality problem that is discovered *after* the software has been released to end-users (or to another activity in the software process). In earlier chapters, we used the term *error* to depict a quality problem that is discovered by software engineers (or others) before the software is released to the end-user (or to another activity in the software process).

The primary objective of formal technical reviews is to find errors during the process so that they do not become defects after release of the software. The obvious benefit of formal technical reviews is the early discovery of errors so that they do not propagate to the next step in the software process.

The primary objective of an FTR is to find errors before they are passed on to another software engineering activity or released to the customer.

A number of industry studies (by TRW, Nippon Electric, Mitre Corp., among others) indicate that design activities introduce between 50 and 65 percent of all errors (and ultimately, all defects) during the software process. However, formal review techniques have been shown to be up to 75 percent effective [JON86] in uncovering design flaws. By detecting and removing a large percentage of these errors, the review process substantially reduces the cost of subsequent steps in the development and support phases.

To illustrate the cost impact of early error detection, we consider a series of relative costs that are based on actual cost data collected for large software projects [IBM81].[3] Assume that an error uncovered during design will cost 1.0 monetary unit to correct. Relative to this cost, the same error uncovered just before testing commences will cost 6.5 units; during testing, 15 units; and after release, between 60 and 100 units.

---

3 Although these data are more than 20 years old, they remain applicable in a modern context.

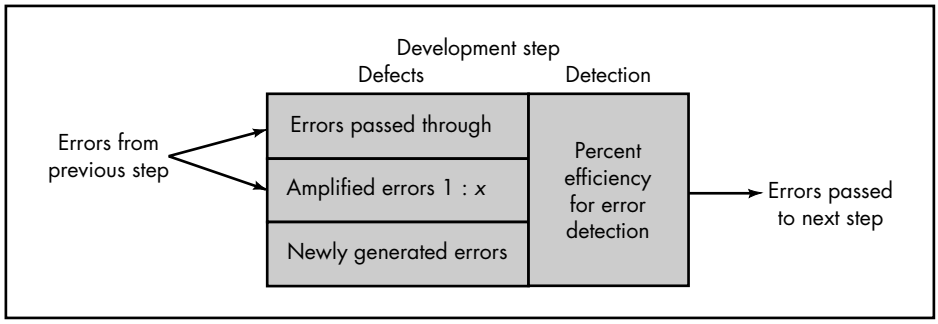


FIGURE 8.2 Defect amplification model

8.4.2 Defect Amplification and Removal

Quote:

"Some maladies, as doctors say, at their beginning are easy to cure but difficult to recognize . . . but in the course of time when they have not at first been recognized and treated, become easy to recognize but difficult to cure."

Niccolo Machiavelli

A defect amplification model [IBM81] can be used to illustrate the generation and detection of errors during the preliminary design, detail design, and coding steps of the software engineering process. The model is illustrated schematically in Figure 8.2. A box represents a software development step. During the step, errors may be inadvertently generated. Review may fail to uncover newly generated errors and errors from previous steps, resulting in some number of errors that are passed through. In some cases, errors passed through from previous steps are amplified (amplification factor, *x*) by current work. The box subdivisions represent each of these characteristics and the percent of efficiency for detecting errors, a function of the thoroughness of the review.

Figure 8.3 illustrates a hypothetical example of defect amplification for a software development process in which no reviews are conducted. Referring to the figure, each test step is assumed to uncover and correct 50 percent of all incoming errors without introducing any new errors (an optimistic assumption). Ten preliminary design defects are amplified to 94 errors before testing commences. Twelve latent errors are released to the field. Figure 8.4 considers the same conditions except that design and code reviews are conducted as part of each development step. In this case, ten initial preliminary design errors are amplified to 24 errors before testing commences. Only three latent errors exist. Recalling the relative costs associated with the discovery and correction of errors, overall cost (with and without review for our hypothetical example) can be established. The number of errors uncovered during each of the steps noted in Figures 8.3 and 8.4 is multiplied by the cost to remove an error (1.5 cost units for design, 6.5 cost units before test, 15 cost units during test, and 67 cost units after release). Using these data, the total cost for development and maintenance when reviews are conducted is 783 cost units. When no reviews are conducted, total cost is 2177 units—nearly three times more costly.

To conduct reviews, a software engineer must expend time and effort and the development organization must spend money. However, the results of the preceding example leave little doubt that we can pay now or pay much more later. Formal tech-

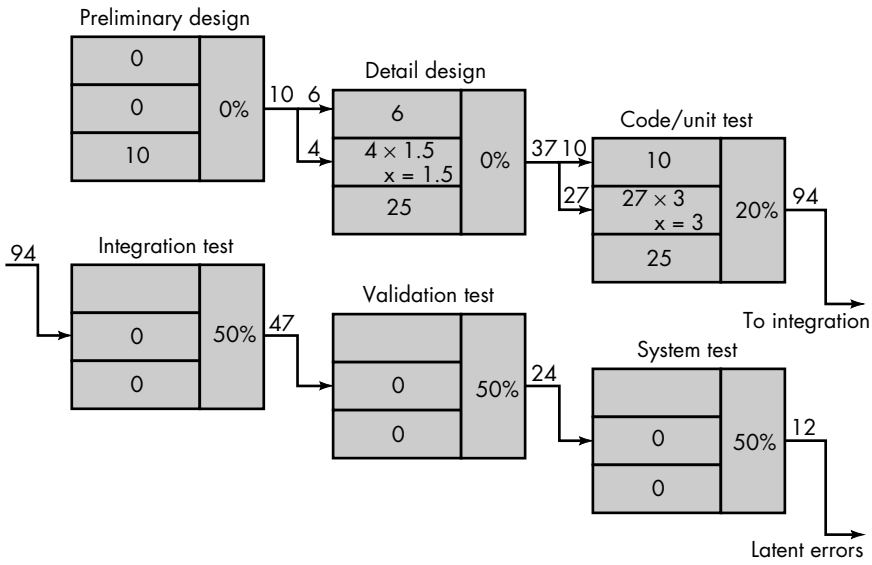


FIGURE 8.3
Defect amplification, no reviews

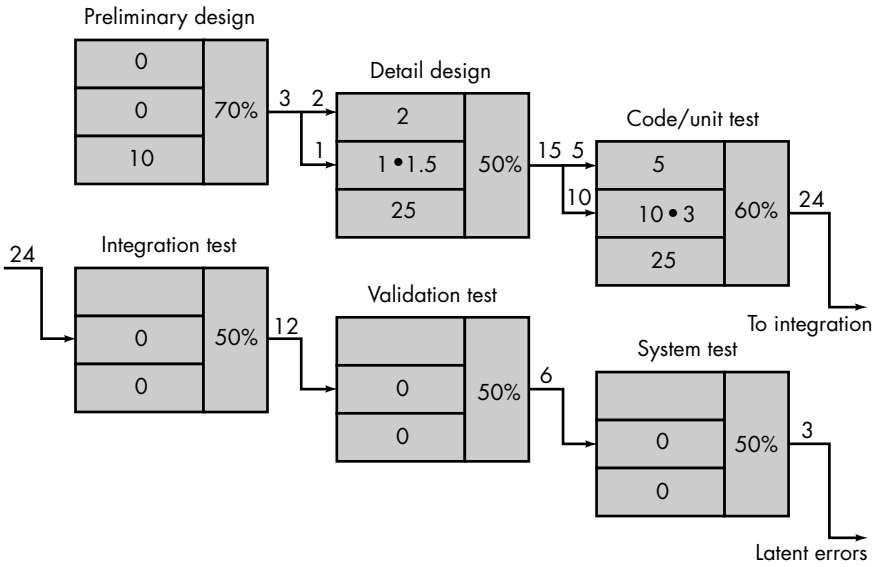


FIGURE 8.4
Defect amplification, reviews conducted

nical reviews (for design and other technical activities) provide a demonstrable cost benefit. They should be conducted.

8.5 FORMAL TECHNICAL REVIEWS

A formal technical review is a software quality assurance activity performed by software engineers (and others). The objectives of the FTR are (1) to uncover errors in function, logic, or implementation for any representation of the software; (2) to verify

When we conduct FTRs, what are our objectives?

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, inspections, round-robin reviews and other small group technical assessments of software. 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 [FRE90], [GIL93] are presented as a representative formal technical review.

8.5.1 The Review Meeting

Quote:

"A meeting is too often an event where minutes are taken and hours are wasted."
author unknown

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 focus, the FTR has a higher likelihood of uncovering errors.

KEY POINT

The FTR focuses on a relatively small portion of a work product.

The focus of the FTR is on a work product (e.g., a portion of a requirements specification, a detailed component design, a source code listing 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.

WebRef

The NASA SATC *Formal Inspection Guidebook* can be downloaded from **satc.gsfc.nasa.gov/fi/fipage.html**

The review meeting is attended by the review leader, all reviewers, and the producer. One of the reviewers takes on the role of the *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). The decision made, all FTR attendees complete a sign-off, indicating their participation in the review and their concurrence with the review team's findings.

8.5.2 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?

Technical Review Summary Report and Issues List

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.

It is important to 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.

8.5.3 Review Guidelines

Guidelines for the conduct of 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 that no review at all. The following represents a minimum set of guidelines for formal technical reviews:

Don't point out errors harshly. One way to be gentle is to ask a question that enables the producer to discover his or her own error.

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.
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 test documents.
8. *Allocate resources and schedule time for FTRs.* For reviews to be effective, they should be scheduled as a task during the software engineering 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.
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.

Quote:

"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

FTR Checklists

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. Porter and his colleagues [POR95] provide excellent guidance for this type of experimentation.

8.6 FORMAL APPROACHES TO SQA

In the preceding sections, we have argued that software quality is everyone's job; that it can be achieved through competent analysis, design, coding, and testing, as well as through the application of formal technical reviews, a multitiered testing strategy, better control of software work products and the changes made to them, and the application of accepted software engineering standards. In addition, quality can be defined in terms of a broad array of quality factors and measured (indirectly) using a variety of indices and metrics.

Over the past two decades, a small, but vocal, segment of the software engineering community has argued that a more formal approach to software quality assurance is required. It can be argued that a computer program is a mathematical object [SOM96]. A rigorous syntax and semantics can be defined for every programming language, and work is underway to develop a similarly rigorous approach to the specification of software requirements. If the requirements model (specification) and the programming language can be represented in a rigorous manner, it should be possible to apply mathematic proof of correctness to demonstrate that a program conforms exactly to its specifications.

XRef

Techniques for formal specification of software are considered in Chapter 25. Correctness proofs are considered in Chapter 26.

Attempts to prove programs correct are not new. Dijkstra [DIJ76] and Linger, Mills, and Witt [LIN79], among others, advocated proofs of program correctness and tied these to the use of structured programming concepts (Chapter 16).

8.7 STATISTICAL SOFTWARE QUALITY ASSURANCE

Statistical quality assurance reflects a growing trend throughout industry to become more quantitative about quality. For software, statistical quality assurance implies the following steps:

? What steps are required to perform statistical SQA?

1. Information about software defects is collected and categorized.
2. An attempt is made to trace each defect to its underlying cause (e.g., nonconformance to specifications, design error, violation of standards, poor communication with the customer).
3. Using the Pareto principle (80 percent of the defects can be traced to 20 percent of all possible causes), isolate the 20 percent (the "vital few").
4. Once the vital few causes have been identified, move to correct the problems that have caused the defects.

Quote:

"20 percent of the code has 80 percent of the defects. Find them, fix them!"

Lowell Arthur

This relatively simple concept represents an important step towards the creation of an adaptive software engineering process in which changes are made to improve those elements of the process that introduce error.

To illustrate this, assume that a software engineering organization collects information on defects for a period of one year. Some of the defects are uncovered as software is being developed. Others are encountered after the software has been released to its end-users. Although hundreds of different errors are uncovered, all can be tracked to one (or more) of the following causes:

- incomplete or erroneous specifications (IES)
- misinterpretation of customer communication (MCC)
- intentional deviation from specifications (IDS)
- violation of programming standards (VPS)
- error in data representation (EDR)
- inconsistent component interface (ICI)
- error in design logic (EDL)
- incomplete or erroneous testing (IET)
- inaccurate or incomplete documentation (IID)
- error in programming language translation of design (PLT)
- ambiguous or inconsistent human/computer interface (HCI)
- miscellaneous (MIS)

WebRef

The Chinese Association for Software Quality presents one of the most comprehensive quality Web sites at **www.casq.org**

To apply statistical SQA, Table 8.1 is built. The table indicates that IES, MCC, and EDR are the *vital few* causes that account for 53 percent of all errors. It should be noted, however, that IES, EDR, PLT, and EDL would be selected as the vital few causes if only serious errors are considered. Once the vital few causes are determined, the software engineering organization can begin corrective action. For example, to correct MCC, the software developer might implement facilitated application specification techniques (Chapter 11) to improve the quality of customer communication and specifications. To improve EDR, the developer might acquire CASE tools for data modeling and perform more stringent data design reviews.

It is important to note that corrective action focuses primarily on the vital few. As the vital few causes are corrected, new candidates pop to the top of the stack.

Statistical quality assurance techniques for software have been shown to provide substantial quality improvement [ART97]. In some cases, software organizations have achieved a 50 percent reduction per year in defects after applying these techniques.

In conjunction with the collection of defect information, software developers can calculate an *error index* (EI) for each major step in the software process [IEE94]. After analysis, design, coding, testing, and release, the following data are gathered:

E_i = the total number of errors uncovered during the ith step in the software engineering process

TABLE 8.1 DATA COLLECTION FOR STATISTICAL SQA

	Total		Serious		Moderate		Minor	
Error	**No.**	**%**	**No.**	**%**	**No.**	**%**	**No.**	**%**
IES	205	22%	34	27%	68	18%	103	24%
MCC	156	17%	12	9%	68	18%	76	17%
IDS	48	5%	1	1%	24	6%	23	5%
VPS	25	3%	0	0%	15	4%	10	2%
EDR	130	14%	26	20%	68	18%	36	8%
ICI	58	6%	9	7%	18	5%	31	7%
EDL	45	5%	14	11%	12	3%	19	4%
IET	95	10%	12	9%	35	9%	48	11%
IID	36	4%	2	2%	20	5%	14	3%
PLT	60	6%	15	12%	19	5%	26	6%
HCI	28	3%	3	2%	17	4%	8	2%
MIS	56	6%	0	0%	15	4%	41	9%
Totals	942	100%	128	100%	379	100%	435	100%

S_i = the number of serious errors
M_i = the number of moderate errors
T_i = the number of minor errors
PS = size of the product (LOC, design statements, pages of documentation) at the ith step
w_s, w_m, w_t = weighting factors for serious, moderate, and trivial errors, where recommended values are $w_s = 10$, $w_m = 3$, $w_t = 1$. The weighting factors for each phase should become larger as development progresses. This rewards an organization that finds errors early.

At each step in the software process, a *phase index,* PI_i, is computed:

$$PI_i = w_s (S_i/E_i) + w_m (M_i/E_i) + w_t (T_i/E_i)$$

The *error index* is computed by calculating the cumulative effect on each PI_i, weighting errors encountered later in the software engineering process more heavily than those encountered earlier:

$$EI = \Sigma(i \times PI_i)/PS$$
$$= (PI_1 + 2PI_2 + 3PI_3 + \ldots iPI_i)/PS$$

The error index can be used in conjunction with information collected in Table 8.1 to develop an overall indication of improvement in software quality.

The application of the statistical SQA and the Pareto principle can be summarized in a single sentence: *Spend your time focusing on things that really matter, but first be sure that you understand what really matters!*

A comprehensive discussion of statistical SQA is beyond the scope of this book. Interested readers should see [SCH98], [KAP95], or [KAN95].

8.8 SOFTWARE RELIABILITY

There is no doubt that the reliability of a computer program is an important element of its overall quality. If a program repeatedly and frequently fails to perform, it matters little whether other software quality factors are acceptable.

The Reliability Analysis Center provides much useful information on reliability, maintainability, supportability, and quality at **rac.iitri.org**

Software reliability, unlike many other quality factors, can be measured directed and estimated using historical and developmental data. *Software reliability* is defined in statistical terms as "the probability of failure-free operation of a computer program in a specified environment for a specified time" [MUS87]. To illustrate, program X is estimated to have a reliability of 0.96 over eight elapsed processing hours. In other words, if program X were to be executed 100 times and require eight hours of elapsed processing time (execution time), it is likely to operate correctly (without failure) 96 times out of 100.

Whenever software reliability is discussed, a pivotal question arises: What is meant by the term *failure?* In the context of any discussion of software quality and reliability, failure is nonconformance to software requirements. Yet, even within this definition, there are gradations. Failures can be only annoying or catastrophic. One failure can be corrected within seconds while another requires weeks or even months to correct. Complicating the issue even further, the correction of one failure may in fact result in the introduction of other errors that ultimately result in other failures.

8.8.1 Measures of Reliability and Availability

Software reliability problems can almost always be traced to errors in design or implementation.

Early work in software reliability attempted to extrapolate the mathematics of hardware reliability theory (e.g., [ALV64]) to the prediction of software reliability. Most hardware-related reliability models are predicated on failure due to wear rather than failure due to design defects. In hardware, failures due to physical wear (e.g., the effects of temperature, corrosion, shock) are more likely than a design-related failure. Unfortunately, the opposite is true for software. In fact, all software failures can be traced to design or implementation problems; wear (see Chapter 1) does not enter into the picture.

There has been debate over the relationship between key concepts in hardware reliability and their applicability to software (e.g., [LIT89], [ROO90]). Although an irrefutable link has yet be be established, it is worthwhile to consider a few simple concepts that apply to both system elements.

If we consider a computer-based system, a simple measure of reliability is *mean-time-between-failure* (MTBF), where

$$\text{MTBF} = \text{MTTF} + \text{MTTR}$$

The acronyms MTTF and MTTR are mean-time-to-failure and mean-time-to-repair, respectively.

? Why is MTBF a more useful metric than defects/KLOC?

Many researchers argue that MTBF is a far more useful measure than defects/KLOC or defects/FP. Stated simply, an end-user is concerned with failures, not with the total error count. Because each error contained within a program does not have the same failure rate, the total error count provides little indication of the reliability of a system. For example, consider a program that has been in operation for 14 months. Many errors in this program may remain undetected for decades before they are discovered. The MTBF of such obscure errors might be 50 or even 100 years. Other errors, as yet undiscovered, might have a failure rate of 18 or 24 months. Even if every one of the first category of errors (those with long MTBF) is removed, the impact on software reliability is negligible.

In addition to a reliability measure, we must develop a measure of availability. *Software availability* is the probability that a program is operating according to requirements at a given point in time and is defined as

$$\text{Availability} = [\text{MTTF}/(\text{MTTF} + \text{MTTR})] \times 100\%$$

The MTBF reliability measure is equally sensitive to MTTF and MTTR. The availability measure is somewhat more sensitive to MTTR, an indirect measure of the maintainability of software.

8.8.2 Software Safety

Leveson [LEV86] discusses the impact of software in safety critical systems when she writes:

> Before software was used in safety critical systems, they were often controlled by conventional (nonprogrammable) mechanical and electronic devices. System safety techniques are designed to cope with random failures in these [nonprogrammable] systems. Human design errors are not considered since it is assumed that all faults caused by human errors can be avoided completely or removed prior to delivery and operation.

Quote:

"I cannot imagine any condition which would cause this ship to founder. Modern shipbuilding has gone beyond that."

E.I. Smith, captain of the Titanic

When software is used as part of the control system, complexity can increase by an order of magnitude or more. Subtle design faults induced by human error—something that can be uncovered and eliminated in hardware-based conventional control—become much more difficult to uncover when software is used.

Software safety is a software quality assurance activity that focuses on the identification and assessment of potential hazards that may affect software negatively and cause an entire system to fail. If hazards can be identified early in the software engineering process, software design features can be specified that will either eliminate or control potential hazards.

A modeling and analysis process is conducted as part of software safety. Initially, hazards are identified and categorized by criticality and risk. For example, some of the hazards associated with a computer-based cruise control for an automobile might be

- causes uncontrolled acceleration that cannot be stopped
- does not respond to depression of brake pedal (by turning off)

- does not engage when switch is activated
- slowly loses or gains speed

Worthwhile papers on software safety (and a detailed glossary) can be found at **www.rstcorp.com/hotlist/topics-safety.html**

Once these system-level hazards are identified, analysis techniques are used to assign severity and probability of occurrence.[4] To be effective, software must be analyzed in the context of the entire system. For example, a subtle user input error (people are system components) may be magnified by a software fault to produce control data that improperly positions a mechanical device. If a set of external environmental conditions are met (and only if they are met), the improper position of the mechanical device will cause a disastrous failure. Analysis techniques such as *fault tree analysis* [VES81], *real-time logic* [JAN86], or *petri net models* [LEV87] can be used to predict the chain of events that can cause hazards and the probability that each of the events will occur to create the chain.

Once hazards are identified and analyzed, safety-related requirements can be specified for the software. That is, the specification can contain a list of undesirable events and the desired system responses to these events. The role of software in managing undesirable events is then indicated.

? What is the difference between software reliability and software safety?

Although software reliability and software safety are closely related to one another, it is important to understand the subtle difference between them. Software reliability uses statistical analysis to determine the likelihood that a software failure will occur. However, the occurrence of a failure does not necessarily result in a hazard or mishap. Software safety examines the ways in which failures result in conditions that can lead to a mishap. That is, failures are not considered in a vacuum, but are evaluated in the context of an entire computer-based system.

A comprehensive discussion of software safety is beyond the scope of this book. Those readers with further interest should refer to Leveson's [LEV95] book on the subject.

8.9 MISTAKE-PROOFING FOR SOFTWARE

If William Shakespeare had commented on the modern software engineer's condition, he might have written: "To err is human, to find the error quickly and correct it is divine." In the 1960s, a Japanese industrial engineer, Shigeo Shingo [SHI86], working at Toyota, developed a quality assurance technique that led to the prevention and/or early correction of errors in the manufacturing process. Called *poka-yoke* (mistake-proofing), Shingo's concept makes use of *poka-yoke* devices—mechanisms that lead to (1) the prevention of a potential quality problem before it occurs or (2) the rapid detection of quality problems if they are introduced. We encounter *poka-yoke* devices in our everyday lives (even if we are unaware of the concept). For exam-

4 This approach is analogous to the risk analysis approach described for software project management in Chapter 6. The primary difference is the emphasis on technology issues as opposed to project-related topics.

ple, the ignition switch for an automobile will not work if an automatic transmission is in gear (a *prevention device*); an auto's warning beep will sound if the seat belts are not buckled (a *detection device*).

A comprehensive collection of *poka-yoke* resources can be obtained at **www.campbell.berry.edu/faculty/jgrout/pokayoke.shtml**

An effective *poka-yoke* device exhibits a set of common characteristics:

- **It is simple and cheap.** If a device is too complicated or expensive, it will not be cost effective.
- **It is part of the process.** That is, the *poka-yoke* device is integrated into an engineering activity.
- **It is located near the process task where the mistakes occur.** Thus, it provides rapid feedback and error correction.

Although *poka-yoke* was originally developed for use in "zero quality control" [SHI86] for manufactured hardware, it can be adapted for use in software engineering. To illustrate, we consider the following problem [ROB97]:

A software products company sells application software to an international market. The pull-down menus and associated mnemonics provided with each application must reflect the local language. For example, the English language menu item for "Close" has the mnemonic "C" associated with it. When the application is sold in a French-speaking country, the same menu item is "Fermer" with the mnemonic "F." To implement the appropriate menu entry for each locale, a "localizer" (a person conversant in the local language and terminology) translates the menus accordingly. The problem is to ensure that (1) each menu entry (there can be hundreds) conforms to appropriate standards and that there are no conflicts, regardless of the language that is used.

The use of *poka-yoke* for testing various application menus implemented in different languages as just described is discussed in a paper by Harry Robinson [ROB97]:[5]

We first decided to break the menu testing problem down into parts that we could solve. Our first advance on the problem was to understand that there were two separate aspects to the message catalogs. There was the content aspect: the simple text translations, such as changing "Close" to "Fermer." Since the test team was not fluent in the 11 target languages, we had to leave this aspect to the language experts.

The second aspect of the message catalogs was the structure, the syntax rules that a properly constructed target catalog must obey. Unlike content, it would be possible for the test team to verify the structural aspects of the catalogs.

As an example of what is meant by structure, consider the labels and mnemonics of an application menu. A menu is made up of labels and associated mnemonics. Each menu, regardless of its contents or its locale, must obey the following rules listed in the Motif Style Guide:

- Each mnemonic must be contained in its associated label
- Each mnemonic must be unique within the menu

5 The paragraphs that follow have been excerpted (with minor editing) from [ROB97] with the permission of the author.

- Each mnemonic must be a single character
- Each mnemonic must be in ASCII

These rules are invariant across locales, and can be used to verify that a menu is constructed correctly in the target locale.

There were several possibilities for how to mistake-proof the menu mnemonics:

Prevention device. We could write a program to generate mnemonics automatically, given a list of the labels in each menu. This approach would prevent mistakes, but the problem of choosing a good mnemonic is difficult and the effort required to write the program would not be justified by the benefit gained.

Prevention device. We could write a program that would prevent the localizer from choosing mnemonics that did not meet the criteria. This approach would also prevent mistakes, but the benefit gained would be minimal; incorrect mnemonics are easy enough to detect and correct after they occur.

Detection device. We could provide a program to verify that the chosen menu labels and mnemonics meet the criteria above. Our localizers could run the programs on their translated message catalogs before sending the catalogs to us. This approach would provide very quick feedback on mistakes, and it is likely as a future step.

Detection device. We could write a program to verify the menu labels and mnemonics, and run the program on message catalogs after they are returned to us by the localizers. This approach is the path we are currently taking. It is not as efficient as some of the above methods, and it can require communication back and forth with the localizers, but the detected errors are still easy to correct at this point.

Several small poka-yoke scripts were used as poka-yoke devices to validate the structural aspects of the menus. A small poka-yoke script would read the table, retrieve the mnemonics and labels from the message catalog, and compare the retrieved strings against the established criteria noted above.

The poka-yoke scripts were small (roughly 100 lines), easy to write (some were written in under an hour) and easy to run. We ran our poka-yoke scripts against 16 applications in the default English locale plus 11 foreign locales. Each locale contained 100 menus, for a total of 1200 menus. The poka-yoke devices found 311 mistakes in menus and mnemonics. Few of the problems we uncovered were earth-shattering, but in total they would have amounted to a large annoyance in testing and running our localized applications.

This example depicts a *poka-yoke* device that has been integrated into software engineering testing activity. The *poka-yoke* technique can be applied at the design, code, and testing levels and provides an effective quality assurance filter.

8.10 THE ISO 9000 QUALITY STANDARDS[6]

A *quality assurance system* may be defined as the organizational structure, responsibilities, procedures, processes, and resources for implementing quality management

6 This section, written by Michael Stovsky, has been adapted from "Fundamentals of ISO 9000" and "ISO 9001 Standard," workbooks developed for *Essential Software Engineering,* a video curriculum developed by R. S. Pressman & Associates, Inc. Reprinted with permission.

[ANS87]. Quality assurance systems are created to help organizations ensure their products and services satisfy customer expectations by meeting their specifications. These systems cover a wide variety of activities encompassing a product's entire life cycle including planning, controlling, measuring, testing and reporting, and improving quality levels throughout the development and manufacturing process. ISO 9000 describes quality assurance elements in generic terms that can be applied to any business regardless of the products or services offered.

Extensive links to ISO 9000/9001 resources can be found at **www.tantara.ab.ca/iso_list.htm**

The ISO 9000 standards have been adopted by many countries including all members of the European Community, Canada, Mexico, the United States, Australia, New Zealand, and the Pacific Rim. Countries in Latin and South America have also shown interest in the standards.

After adopting the standards, a country typically permits only ISO registered companies to supply goods and services to government agencies and public utilities. Telecommunication equipment and medical devices are examples of product categories that must be supplied by ISO registered companies. In turn, manufacturers of these products often require their suppliers to become registered. Private companies such as automobile and computer manufacturers frequently require their suppliers to be ISO registered as well.

To become registered to one of the quality assurance system models contained in ISO 9000, a company's quality system and operations are scrutinized by third party auditors for compliance to the standard and for effective operation. Upon successful registration, a company is issued a certificate from a registration body represented by the auditors. Semi-annual surveillance audits ensure continued compliance to the standard.

8.10.1 The ISO Approach to Quality Assurance Systems

The ISO 9000 quality assurance models treat an enterprise as a network of interconnected processes. For a quality system to be ISO compliant, these processes must address the areas identified in the standard and must be documented and practiced as described.

ISO 9000 describes what must be done to be compliant, but it does not describe how it must be done.

ISO 9000 describes the elements of a quality assurance system in general terms. These elements include the organizational structure, procedures, processes, and resources needed to implement quality planning, quality control, quality assurance, and quality improvement. However, ISO 9000 does not describe how an organization should implement these quality system elements. Consequently, the challenge lies in designing and implementing a quality assurance system that meets the standard and fits the company's products, services, and culture.

8.10.2 The ISO 9001 Standard

ISO 9001 is the quality assurance standard that applies to software engineering. The standard contains 20 requirements that must be present for an effective quality assurance system. Because the ISO 9001 standard is applicable to all engineering

disciplines, a special set of ISO guidelines (ISO 9000-3) have been developed to help interpret the standard for use in the software process.

ISO 9000 for Software

The requirements delineated by ISO 9001 address topics such as management responsibility, quality system, contract review, design control, document and data control, product identification and traceability, process control, inspection and testing, corrective and preventive action, control of quality records, internal quality audits, training, servicing, and statistical techniques. In order for a software organization to become registered to ISO 9001, it must establish policies and procedures to address each of the requirements just noted (and others) and then be able to demonstrate that these policies and procedures are being followed. For further information on ISO 9001, the interested reader should see [HOY98], [SCH97], or [SCH94].

8.11 THE SQA PLAN

The *SQA Plan* provides a road map for instituting software quality assurance. Developed by the SQA group, the plan serves as a template for SQA activities that are instituted for each software project.

The SQA plan

A standard for SQA plans has been recommended by the IEEE [IEE94]. Initial sections describe the purpose and scope of the document and indicate those software process activities that are covered by quality assurance. All documents noted in the *SQA Plan* are listed and all applicable standards are noted. The management section of the plan describes SQA's place in the organizational structure, SQA tasks and activities and their placement throughout the software process, and the organizational roles and responsibilities relative to product quality.

The documentation section describes (by reference) each of the work products produced as part of the software process. These include

- project documents (e.g., project plan)
- models (e.g., ERDs, class hierarchies)
- technical documents (e.g., specifications, test plans)
- user documents (e.g., help files)

In addition, this section defines the minimum set of work products that are acceptable to achieve high quality.

The standards, practices, and conventions section lists all applicable standards and practices that are applied during the software process (e.g., document standards, coding standards, and review guidelines). In addition, all project, process, and (in some instances) product metrics that are to be collected as part of software engineering work are listed.

The reviews and audits section of the plan identifies the reviews and audits to be conducted by the software engineering team, the SQA group, and the customer. It provides an overview of the approach for each review and audit.

The test section references the *Software Test Plan and Procedure* (Chapter 18). It also defines test record-keeping requirements. Problem reporting and corrective action defines procedures for reporting, tracking, and resolving errors and defects, and identifies the organizational responsibilities for these activities.

The remainder of the *SQA Plan* identifies the tools and methods that support SQA activities and tasks; references software configuration management procedures for controlling change; defines a contract management approach; establishes methods for assembling, safeguarding, and maintaining all records; identifies training required to meet the needs of the plan; and defines methods for identifying, assessing, monitoring, and controlling risk.

8.12 SUMMARY

Software quality assurance is an umbrella activity that is applied at each step in the software process. SQA encompasses procedures for the effective application of methods and tools, formal technical reviews, testing strategies and techniques, *poka-yoke* devices, procedures for change control, procedures for assuring compliance to standards, and measurement and reporting mechanisms.

SQA is complicated by the complex nature of software quality—an attribute of computer programs that is defined as "conformance to explicitly and implicitly specified requirements." But when considered more generally, software quality encompasses many different product and process factors and related metrics.

Software reviews are one of the most important SQA activities. Reviews serve as filters throughout all software engineering activities, removing errors while they are relatively inexpensive to find and correct. The formal technical review is a stylized meeting that has been shown to be extremely effective in uncovering errors.

To properly conduct software quality assurance, data about the software engineering process should be collected, evaluated, and disseminated. Statistical SQA helps to improve the quality of the product and the software process itself. Software reliability models extend measurements, enabling collected defect data to be extrapolated into projected failure rates and reliability predictions.

In summary, we recall the words of Dunn and Ullman [DUN82]: "Software quality assurance is the mapping of the managerial precepts and design disciplines of quality assurance onto the applicable managerial and technological space of software engineering." The ability to ensure quality is the measure of a mature engineering discipline. When the mapping is successfully accomplished, mature software engineering is the result.

REFERENCES

[ALV64] Alvin, W.H. von (ed.), *Reliability Engineering,* Prentice-Hall, 1964.

[ANS87] ANSI/ASQC A3-1987, Quality Systems Terminology, 1987.

[ART92] Arthur, L.J., *Improving Software Quality: An Insider's Guide to TQM,* Wiley, 1992.

[ART97] Arthur, L.J., "Quantum Improvements in Software System Quality, *CACM,* vol. 40, no. 6, June 1997, pp. 47–52.

[BOE81] Boehm, B., *Software Engineering Economics,* Prentice-Hall, 1981.

[CRO79] Crosby, P., *Quality Is Free,* McGraw-Hill, 1979.

[DEM86] Deming, W.E., *Out of the Crisis,* MIT Press, 1986.

[DEM99] DeMarco, T., "Management Can Make Quality (Im)possible," *Cutter IT Summit,* Boston, April 1999.

[DIJ76] Dijkstra, E., *A Discipline of Programming,* Prentice-Hall, 1976.

[DUN82] Dunn, R. and R. Ullman, *Quality Assurance for Computer Software,* McGraw-Hill, 1982.

[FRE90] Freedman, D.P. and G.M. Weinberg, *Handbook of Walkthroughs, Inspections and Technical Reviews,* 3rd ed., Dorset House, 1990.

[GIL93] Gilb, T. and D. Graham, *Software Inspections,* Addison-Wesley, 1993.

[GLA98] Glass, R., "Defining Quality Intuitively," *IEEE Software,* May 1998, pp. 103–104, 107.

[HOY98] Hoyle, D., *ISO 9000 Quality Systems Development Handbook: A Systems Engineering Approach,* Butterworth-Heinemann, 1998.

[IBM81] "Implementing Software Inspections," course notes, IBM Systems Sciences Institute, IBM Corporation, 1981.

IEE94] *Software Engineering Standards,* 1994 ed., IEEE Computer Society, 1994.

[JAN86] Jahanian, F. and A.K. Mok, "Safety Analysis of Timing Properties of Real-Time Systems", *IEEE Trans. Software Engineering,* vol. SE-12, no. 9, September 1986, pp. 890–904.

[JON86] Jones, T.C., *Programming Productivity,* McGraw-Hill, 1986.

[KAN95] Kan, S.H., *Metrics and Models in Software Quality Engineering,* Addison-Wesley, 1995.

[KAP95] Kaplan, C., R. Clark, and V. Tang, *Secrets of Software Quality: 40 Innovations from IBM,* McGraw-Hill, 1995.

[LEV86] Leveson, N.G., "Software Safety: Why, What, and How," *ACM Computing Surveys,* vol. 18, no. 2, June 1986, pp. 125–163.

[LEV87] Leveson, N.G. and J.L. Stolzy, "Safety Analysis Using Petri Nets, *IEEE Trans. Software Engineering,* vol. SE-13, no. 3, March 1987, pp. 386–397.

[LEV95] Leveson, N.G., *Safeware: System Safety And Computers,* Addison-Wesley, 1995.

[LIN79] Linger, R., H. Mills, and B. Witt, *Structured Programming,* Addison-Wesley, 1979.

[LIT89] Littlewood, B., "Forecasting Software Reliability," in *Software Reliability: Modeling and Identification,* (S. Bittanti, ed.), Springer-Verlag, 1989, pp. 141–209.
[MUS87] Musa, J.D., A. Iannino, and K. Okumoto, *Engineering and Managing Software with Reliability Measures,* McGraw-Hill, 1987.
[POR95] Porter, A., H. Siy, C.A. Toman, and L.G. Votta, "An Experiment to Assess the Cost-Benefits of Code Inspections in Large Scale Software Development," Proc. *Third ACM SIGSOFT Symposium on the Foundations of Software Engineering,* Washington, D.C., October 1996, ACM Press, pp. 92–103.
[ROB97] Robinson, H., "Using Poka-Yoke Techniques for Early Error Detection," *Proc. Sixth International Conference on Software Testing Analysis and Review* (STAR'97), 1997, pp. 119–142.
[ROO90] Rook, J., *Software Reliability Handbook,* Elsevier, 1990.
[SCH98] Schulmeyer, G.C. and J.I. McManus (eds.), *Handbook of Software Quality Assurance,* 3rd ed., Prentice-Hall, 1998.
[SCH94] Schmauch, C.H., *ISO 9000 for Software Developers,* ASQC Quality Press, 1994.
[SCH97] Schoonmaker, S.J., *ISO 9001 for Engineers and Designers,* McGraw-Hill, 1997.
[SHI86] Shigeo Shingo, *Zero Quality Control: Source Inspection and the Poka-yoke System,* Productivity Press, 1986.
[SOM96] Somerville, I., *Software Engineering,* 5th ed., Addison-Wesley, 1996.
[VES81] Veseley, W.E., et al., *Fault Tree Handbook,* U.S. Nuclear Regulatory Commission, NUREG-0492, January 1981.

PROBLEMS AND POINTS TO PONDER

8.1. Early in this chapter we noted that "variation control is the heart of quality control." Since every program that is created is different from every other program, what are the variations that we look for and how do we control them?

8.2. Is it possible to assess the quality of software if the customer keeps changing what it is supposed to do?

8.3. Quality and reliability are related concepts but are fundamentally different in a number of ways. Discuss them.

8.4. Can a program be correct and still not be reliable? Explain.

8.5. Can a program be correct and still not exhibit good quality? Explain.

8.6. Why is there often tension between a software engineering group and an independent software quality assurance group? Is this healthy?

8.7. You have been given the responsibility for improving the quality of software across your organization. What is the first thing that you should do? What's next?

8.8. Besides counting errors, are there other countable characteristics of software that imply quality? What are they and can they be measured directly?

8.9. A formal technical review is effective only if everyone has prepared in advance. How do you recognize a review participant who has not prepared? What do you do if you're the review leader?

8.10. Some people argue that an FTR should assess programming style as well as correctness. Is this a good idea? Why?

8.11. Review Table 8.1 and select four vital few causes of serious and moderate errors. Suggest corrective actions using information presented in other chapters.

8.12. An organization uses a five-step software engineering process in which errors are found according to the following percentage distribution:

Step	Percentage of errors found
1	20%
2	15%
3	15%
4	40%
5	10%

Using Table 8.1 information and this percentage distribution, compute the overall defect index for the organization. Assume PS = 100,000.

8.13. Research the literature on software reliability and write a paper that describes one software reliability model. Be sure to provide an example.

8.14. The MTBF concept for software is open to criticism. Can you think of a few reasons why?

8.15. Consider two safety critical systems that are controlled by computer. List at least three hazards for each that can be directly linked to software failures.

8.16. Using Web and print resources, develop a 20 minute tutorial on *poka-yoke* and present it to your class.

8.17. Suggest a few *poka-yoke* devices that might be used to detect and/or prevent errors that are commonly encountered prior to "sending" an e-mail message.

8.18. Acquire a copy of ISO 9001 and ISO 9000-3. Prepare a presentation that discusses three ISO 9001 requirements and how they apply in a software context.

FURTHER READINGS AND INFORMATION SOURCES

Books by Moriguchi (*Software Excellence: A Total Quality Management Guide,* Productivity Press, 1997) and Horch (*Practical Guide to Software Quality Management,* Artech Publishing, 1996) are excellent management-level presentations on the benefits of formal quality assurance programs for computer software. Books by Deming [DEM86] and Crosby [CRO79] do not focus on software, but both books are must reading for

senior managers with software development responsibility. Gluckman and Roome (*Everyday Heroes of the Quality Movement,* Dorset House, 1993) humanizes quality issues by telling the story of the players in the quality process. Kan *(Metrics and Models in Software Quality Engineering,* Addison-Wesley, 1995) presents a quantitative view of software quality.

Tingley (*Comparing ISO 9000, Malcolm Baldrige, and the SEI CMM for Software,* Prentice-Hall, 1996) provides useful guidance for organizations that are striving to improve their quality management processes. Oskarsson (*An ISO 9000 Approach to Building Quality Software,* Prentice-Hall, 1995) discusses the ISO standard as it applies to software.

Dozens of books have been written about software quality issues in recent years. The following is a small sampling of useful sources:

Clapp, J.A., et al., *Software Quality Control, Error Analysis and Testing,* Noyes Data Corp., 1995.

Dunn, R.H. and R.S. Ullman, *TQM for Computer Software,* McGraw-Hill, 1994.

Fenton, N., R. Whitty, and Y. Iizuka, *Software Quality Assurance and Measurement: Worldwide Industrial Applications,* Chapman & Hall, 1994.

Ferdinand, A.E., *Systems, Software, and Quality Engineering,* Van Nostrand-Reinhold, 1993.

Ginac, F.P., *Customer Oriented Software Quality Assurance,* Prentice-Hall, 1998.

Ince, D. , *ISO 9001 and Software Quality Assurance,* McGraw-Hill, 1994.

Ince, D., *An Introduction to Software Quality Assurance and Its Implementation,* McGraw-Hill, 1994.

Jarvis, A. and V. Crandall, *Inroads to Software Quality: "How to" Guide and Toolkit,* Prentice-Hall, 1997.

Sanders, J., *Software Quality: A Framework for Success in Software Development,* Addison-Wesley, 1994.

Sumner, F.H., *Software Quality Assurance,* Macmillan, 1993.

Wallmuller, E., *Software Quality Assurance: A Practical Approach,* Prentice-Hall, 1995.

Weinberg, G.M., *Quality Software Management,* four volumes, Dorset House, 1992, 1993, 1994, 1996.

Wilson, R.C., *Software Rx: Secrets of Engineering Quality Software,* Prentice-Hall, 1997.

An anthology edited by Wheeler, Brykczynski, and Meeson (*Software Inspection: Industry Best Practice,* IEEE Computer Society Press, 1996) presents useful information on this important SQA activity. Friedman and Voas (*Software Assessment,* Wiley, 1995) discuss both theoretical underpinnings and practical methods for ensuring the reliability and safety of computer programs.

Musa (*Software Reliability Engineering: More Reliable Software, Faster Development and Testing,* McGraw-Hill, 1998) has written a practical guide to applied software reliability techniques. Anthologies of important papers on software reliability have been edited by Kapur et al. (*Contributions to Hardware and Software Reliability Modelling,* World Scientific Publishing Co., 1999), Gritzalis (*Reliability, Quality and Safety of Software-Intensive Systems,* Kluwer Academic Publishers, 1997), and Lyu (*Handbook*

of Software Reliability Engineering, McGraw-Hill, 1996). Storey (*Safety-Critical Computer Systems,* Addison-Wesley, 1996) and Leveson [LEV95] continue to be the most comprehensive discussions of software safety published to date.

In addition to [SHI86], the *poka-yoke* technique for mistake-proofing software is discussed by Shingo (*The Shingo Production Management System: Improving Process Functions,* Productivity Press, 1992) and Shimbun *(Poka-Yoke: Improving Product Quality by Preventing Defects,* Productivity Press, 1989).

A wide variety of information sources on software quality assurance, software reliability, and related subjects is available on the Internet. An up-to-date list of World Wide Web references that are relevant to software quality can be found at the SEPA Web site:

http://www.mhhe.com/engcs/compsci/pressman/resources/sqa.mhtml

CHAPTER

9 SOFTWARE CONFIGURATION MANAGEMENT

KEY CONCEPTS

access control . . . 234
baselines. 227
change control . . 234
configuration audit 237
configuration's objects. 229
identification . . . 230
SCIs 228
SCM process. . . . 230
status reporting. 237
synchronization control. 234
version control. . 232

Change is inevitable when computer software is built. And change increases the level of confusion among software engineers who are working on a project. Confusion arises when changes are not analyzed before they are made, recorded before they are implemented, reported to those with a need to know, or controlled in a manner that will improve quality and reduce error. Babich [BAB86] discusses this when he states:

> The art of coordinating software development to minimize . . . confusion is called *configuration management.* Configuration management is the art of identifying, organizing, and controlling modifications to the software being built by a programming team. The goal is to maximize productivity by minimizing mistakes.

Software configuration management (SCM) is an umbrella activity that is applied throughout the software process. Because change can occur at any time, SCM activities are developed to (1) identify change, (2) control change, (3) ensure that change is being properly implemented, and (4) report changes to others who may have an interest.

It is important to make a clear distinction between software support and software configuration management. Support is a set of software engineering activities that occur after software has been delivered to the customer and put

QUICK LOOK

What is it? When you build computer software, change happens. And because it happens, you need to control it effectively. Software configuration management (SCM) is a set of activities designed to control change by identifying the work products that are likely to change, establishing relationships among them, defining mechanisms for managing different versions of these work products, controlling the changes imposed, and auditing and reporting on the changes made.

Who does it? Everyone involved in the software engineering process is involved with SCM to some extent, but specialized support positions are sometimes created to manage the SCM process.

Why is it important? If you don't control change, it controls you. And that's never good. It's very easy for a stream of uncontrolled changes to turn a well-run software project into chaos. For that reason, SCM is an essential part of good project management and solid software engineering practice.

What are the steps? Because many work products are produced when software is built, each must be uniquely identified. Once this is accomplished, mechanisms for version and change control can be established. To ensure that quality is maintained as changes are made, the process is audited; and to ensure that those with a need to know are informed about changes, reporting is conducted.

QUICK LOOK

What is the work product? The Software Configuration Management Plan defines the project strategy for SCM. In addition, when formal SCM is invoked, the change control process produces software change requests and reports and engineering change orders.

How do I ensure that I've done it right? When every work product can be accounted for, traced, and controlled; when every change can be tracked and analyzed; when everyone who needs to know about a change has been informed—you've done it right.

into operation. Software configuration management is a set of tracking and control activities that begin when a software engineering project begins and terminate only when the software is taken out of operation.

A primary goal of software engineering is to improve the ease with which changes can be accommodated and reduce the amount of effort expended when changes must be made. In this chapter, we discuss the specific activities that enable us to manage change.

9.1 SOFTWARE CONFIGURATION MANAGEMENT

The output of the software process is information that may be divided into three broad categories: (1) computer programs (both source level and executable forms); (2) documents that describe the computer programs (targeted at both technical practitioners and users), and (3) data (contained within the program or external to it). The items that comprise all information produced as part of the software process are collectively called a *software configuration.*

Quote:

"There is nothing permanent except change."

Heraclitus
500 B.C.

As the software process progresses, the number of *software configuration items* (SCIs) grows rapidly. A *System Specification* spawns a *Software Project Plan* and *Software Requirements Specification* (as well as hardware related documents). These in turn spawn other documents to create a hierarchy of information. If each SCI simply spawned other SCIs, little confusion would result. Unfortunately, another variable enters the process—*change.* Change may occur at any time, for any reason. In fact, the First Law of System Engineering [BER80] states: "No matter where you are in the system life cycle, the system will change, and the desire to change it will persist throughout the life cycle."

What is the origin of these changes? The answer to this question is as varied as the changes themselves. However, there are four fundamental sources of change:

- New business or market conditions dictate changes in product requirements or business rules.
- New customer needs demand modification of data produced by information systems, functionality delivered by products, or services delivered by a computer-based system.

- Reorganization or business growth/downsizing causes changes in project priorities or software engineering team structure.
- Budgetary or scheduling constraints cause a redefinition of the system or product.

Software configuration management is a set of activities that have been developed to manage change throughout the life cycle of computer software. SCM can be viewed as a software quality assurance activity that is applied throughout the software process. In the sections that follow, we examine major SCM tasks and important concepts that help us to manage change.

9.1.1 Baselines

Most software changes are justified. Don't bemoan changes. Rather, be certain that you have mechanisms in place to handle them.

Change is a fact of life in software development. Customers want to modify requirements. Developers want to modify the technical approach. Managers want to modify the project strategy. Why all this modification? The answer is really quite simple. As time passes, all constituencies know more (about what they need, which approach would be best, how to get it done and still make money). This additional knowledge is the driving force behind most changes and leads to a statement of fact that is difficult for many software engineering practitioners to accept: Most changes are justified!

A *baseline* is a software configuration management concept that helps us to control change without seriously impeding justifiable change. The IEEE (IEEE Std. No. 610.12-1990) defines a baseline as:

> A specification or product that has been formally reviewed and agreed upon, that thereafter serves as the basis for further development, and that can be changed only through formal change control procedures.

One way to describe a baseline is through analogy:

> Consider the doors to the kitchen in a large restaurant. One door is marked OUT and the other is marked IN. The doors have stops that allow them to be opened only in the appropriate direction.
>
> If a waiter picks up an order in the kitchen, places it on a tray and then realizes he has selected the wrong dish, he may change to the correct dish quickly and informally before he leaves the kitchen.
>
> If, however, he leaves the kitchen, gives the customer the dish and then is informed of his error, he must follow a set procedure: (1) look at the check to determine if an error has occurred, (2) apologize profusely, (3) return to the kitchen through the IN door, (4) explain the problem, and so forth.

A software engineering work product becomes a baseline only after it has been reviewed and approved.

A baseline is analogous to the kitchen doors in the restaurant. Before a software configuration item becomes a baseline, change may be made quickly and informally. However, once a baseline is established, we figuratively pass through a swinging one-way door. Changes can be made, but a specific, formal procedure must be applied to evaluate and verify each change.

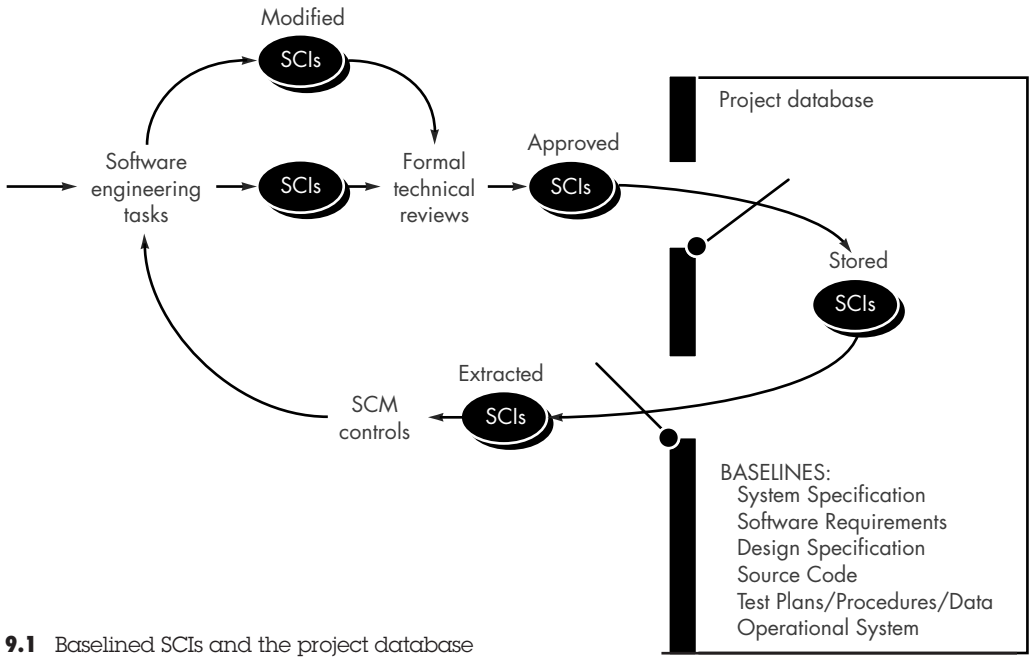


FIGURE 9.1 Baselined SCIs and the project database

In the context of software engineering, a baseline is a milestone in the development of software that is marked by the delivery of one or more software configuration items and the approval of these SCIs that is obtained through a formal technical review (Chapter 8). For example, the elements of a *Design Specification* have been documented and reviewed. Errors are found and corrected. Once all parts of the specification have been reviewed, corrected and then approved, the *Design Specification* becomes a baseline. Further changes to the program architecture (documented in the *Design Specification*) can be made only after each has been evaluated and approved. Although baselines can be defined at any level of detail, the most common software baselines are shown in Figure 9.1.

Be sure that the project database is maintained in a centralized, controlled location.

The progression of events that lead to a baseline is also illustrated in Figure 9.1. Software engineering tasks produce one or more SCIs. After SCIs are reviewed and approved, they are placed in a *project database* (also called a *project library* or *software repository*). When a member of a software engineering team wants to make a modification to a baselined SCI, it is copied from the project database into the engineer's private work space. However, this extracted SCI can be modified only if SCM controls (discussed later in this chapter) are followed. The arrows in Figure 9.1 illustrate the modification path for a baselined SCI.

9.1.2 Software Configuration Items

We have already defined a software configuration item as information that is created as part of the software engineering process. In the extreme, a SCI could be considered to be a single section of a large specification or one test case in a large suite of

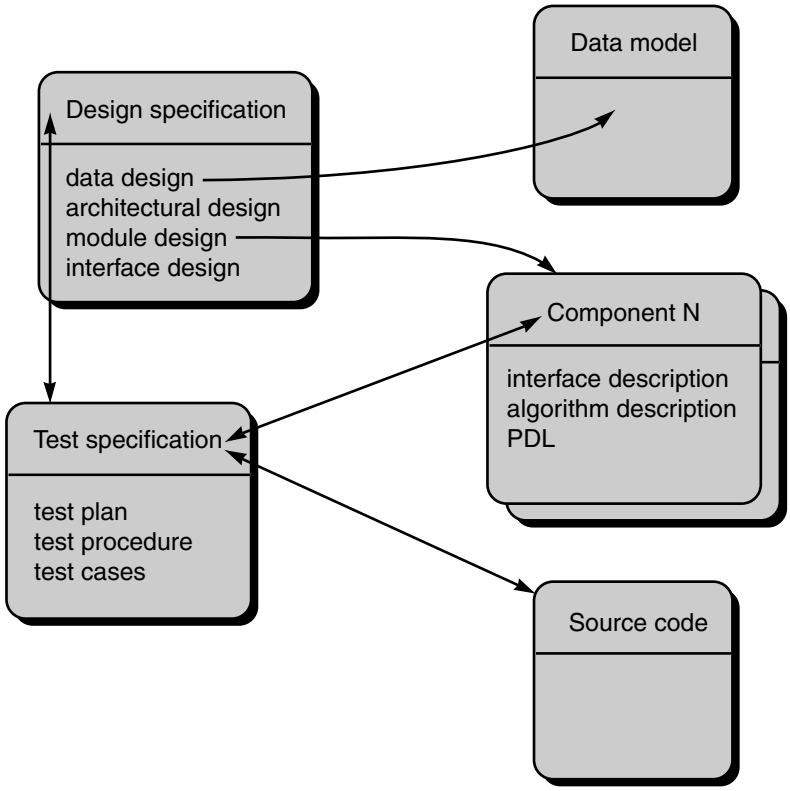


FIGURE 9.2
Configuration objects

tests. More realistically, an SCI is a document, a entire suite of test cases, or a named program component (e.g., a C++ function or an Ada package).

Software Configuration Items

In addition to the SCIs that are derived from software work products, many software engineering organizations also place software tools under configuration control. That is, specific versions of editors, compilers, and other CASE tools are "frozen" as part of the software configuration. Because these tools were used to produce documentation, source code, and data, they must be available when changes to the software configuration are to be made. Although problems are rare, it is possible that a new version of a tool (e.g., a compiler) might produce different results than the original version. For this reason, tools, like the software that they help to produce, can be baselined as part of a comprehensive configuration management process.

In reality, SCIs are organized to form *configuration objects* that may be cataloged in the project database with a single name. A configuration object has a name, attributes, and is "connected" to other objects by relationships. Referring to Figure 9.2, the configuration objects, **Design Specification, data model, component N, source code** and **Test Specification** are each defined separately. However, each of the objects is related to the others as shown by the arrows. A curved arrow indicates a *compositional relation.* That is, **data model** and **component N** are part of the object **Design Specification.** A double-headed straight arrow indicates an interrelationship.

If a change were made to the source code object, the interrelationships enable a software engineer to determine what other objects (and SCIs) might be affected.[1]

9.2 THE SCM PROCESS

Software configuration management is an important element of software quality assurance. Its primary responsibility is the control of change. However, SCM is also responsible for the identification of individual SCIs and various versions of the software, the auditing of the software configuration to ensure that it has been properly developed, and the reporting of all changes applied to the configuration.

Any discussion of SCM introduces a set of complex questions:

- How does an organization identify and manage the many existing versions of a program (and its documentation) in a manner that will enable change to be accommodated efficiently?
- How does an organization control changes before and after software is released to a customer?
- Who has responsibility for approving and ranking changes?
- How can we ensure that changes have been made properly?
- What mechanism is used to appraise others of changes that are made?

WebRef

The *Configuration Management Yellow Pages* contains the most comprehensive listing of SCM resources on the Web at **www.cs.colorado.edu/users/andre/configuration_management.html**

These questions lead us to the definition of five SCM tasks: *identification, version control, change control, configuration auditing,* and *reporting.*

9.3 IDENTIFICATION OF OBJECTS IN THE SOFTWARE CONFIGURATION

To control and manage software configuration items, each must be separately named and then organized using an object-oriented approach. Two types of objects can be identified [CHO89]: *basic objects* and *aggregate objects.*[2] A basic object is a "unit of text" that has been created by a software engineer during analysis, design, code, or test. For example, a basic object might be a section of a requirements specification, a source listing for a component, or a suite of test cases that are used to exercise the code. An aggregate object is a collection of basic objects and other aggregate objects. Referring to Figure 9.2, **Design Specification** is an aggregate object. Conceptually, it can be viewed as a named (identified) list of pointers that specify basic objects such as **data model** and **component N**.

Each object has a set of distinct features that identify it uniquely: a name, a description, a list of resources, and a "realization." The object name is a character string that identifies the object unambiguously. The object description is a list of data items that identify

1 These relationships are defined within the database. The structure of the project database will be discussed in greater detail in Chapter 31.

2 The concept of an aggregate object [GUS89] has been proposed as a mechanism for representing a complete version of a software configuration.

- the SCI type (e.g., document, program, data) represented by the object
- a project identifier
- change and/or version information

KEY POINT

The interrelationships established for configuration objects allow a software engineer to assess the impact of change.

Resources are "entities that are provided, processed, referenced or otherwise required by the object [CHO89]." For example, data types, specific functions, or even variable names may be considered to be object resources. The realization is a pointer to the "unit of text" for a basic object and null for an aggregate object.

Configuration object identification must also consider the relationships that exist between named objects. An object can be identified as **<part-of>** an aggregate object. The relationship **<part-of>** defines a hierarchy of objects. For example, using the simple notation

E-R diagram 1.4 <part-of> data model;
data model <part-of> design specification;

we create a hierarchy of SCIs.

XRef

Data models and data flow diagrams are discussed in Chapter 12.

It is unrealistic to assume that the only relationships among objects in an object hierarchy are along direct paths of the hierarchical tree. In many cases, objects are interrelated across branches of the object hierarchy. For example, a data model is interrelated to data flow diagrams (assuming the use of structured analysis) and also interrelated to a set of test cases for a specific equivalence class. These cross structural relationships can be represented in the following manner:

data model <interrelated> data flow model;
data model <interrelated> test case class m;

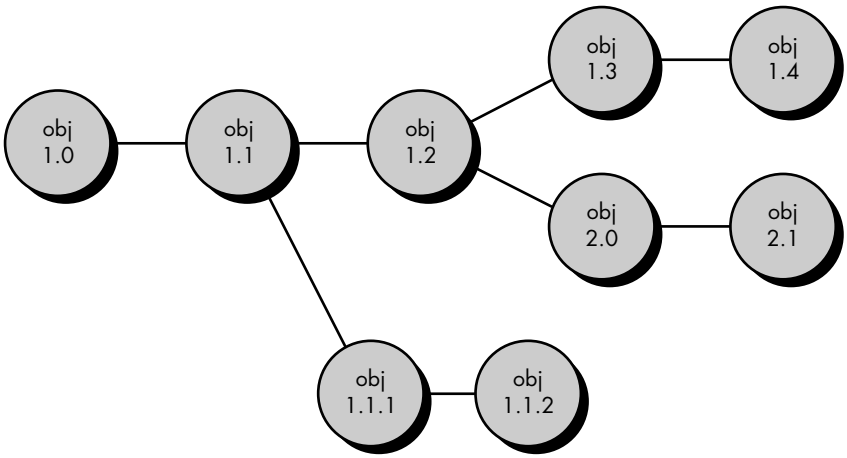
In the first case, the interrelationship is between a composite object, while the second relationship is between an aggregate object (**data model**) and a basic object (**test case class m**).

The interrelationships between configuration objects can be represented with a *module interconnection language* (MIL) [NAR87]. A MIL describes the interdependencies among configuration objects and enables any version of a system to be constructed automatically.

The identification scheme for software objects must recognize that objects evolve throughout the software process. Before an object is baselined, it may change many times, and even after a baseline has been established, changes may be quite frequent. It is possible to create an *evolution graph* [GUS89] for any object. The evolution graph describes the change history of an object, as illustrated in Figure 9.3. Configuration object 1.0 undergoes revision and becomes object 1.1. Minor corrections and changes result in versions 1.1.1 and 1.1.2, which is followed by a major update that is object 1.2. The evolution of object 1.0 continues through 1.3 and 1.4, but at the same time, a major modification to the object results in a new evolutionary path, version 2.0. Both versions are currently supported.

Changes may be made to any version, but not necessarily to all versions. How does the developer reference all components, documents, and test cases for version 1.4? How does the marketing department know what customers currently have

FIGURE 9.3
Evolution graph



version 2.1? How can we be sure that changes to the version 2.1 source code are properly reflected in the corresponding design documentation? A key element in the answer to all these questions is identification.

A variety of automated SCM tools has been developed to aid in identification (and other SCM) tasks. In some cases, a tool is designed to maintain full copies of only the most recent version. To achieve earlier versions (of documents or programs) changes (cataloged by the tool) are "subtracted" from the most recent version [TIC82]. This scheme makes the current configuration immediately available and allows other versions to be derived easily.

CASE Tools—SCM

9.4 VERSION CONTROL

Version control combines procedures and tools to manage different versions of configuration objects that are created during the software process. Clemm [CLE89] describes version control in the context of SCM:

> Configuration management allows a user to specify alternative configurations of the software system through the selection of appropriate versions. This is supported by associating attributes with each software version, and then allowing a configuration to be specified [and constructed] by describing the set of desired attributes.

The naming scheme you establish for SCIs should incorporate the version number.

These "attributes" mentioned can be as simple as a specific version number that is attached to each object or as complex as a string of Boolean variables (switches) that indicate specific types of functional changes that have been applied to the system [LIE89].

One representation of the different versions of a system is the evolution graph presented in Figure 9.3. Each node on the graph is an aggregate object, that is, a complete version of the software. Each version of the software is a collection of SCIs (source code, documents, data), and each version may be composed of different variants. To illustrate this concept, consider a version of a simple program that is com-

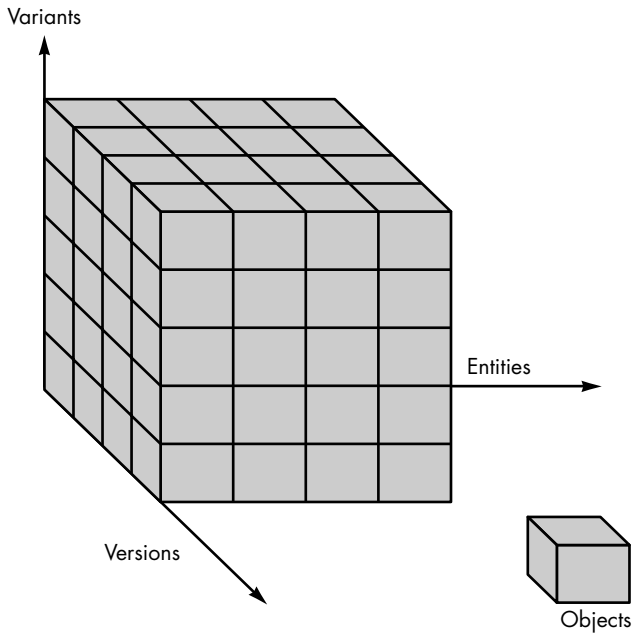


FIGURE 9.4
Object pool representation of components, variants, and versions [REI89]

posed of entities 1, 2, 3, 4, and 5.[3] Entity 4 is used only when the software is implemented using color displays. Entity 5 is implemented when monochrome displays are available. Therefore, two variants of the version can be defined: (1) entities 1, 2, 3, and 4; (2) entities 1, 2, 3, and 5.

Quote:
"Any change, even a change for the better, is always accompanied by drawbacks and discomforts."
Arnold Bennett

To construct the appropriate *variant* of a given version of a program, each entity can be assigned an "attribute-tuple"—a list of features that will define whether the entity should be used when a particular variant of a software version is to be constructed. One or more attributes is assigned for each variant. For example, a color attribute could be used to define which entity should be included when color displays are to be supported.

Another way to conceptualize the relationship between entities, variants and versions (revisions) is to represent them as an *object pool* [REI89]. Referring to Figure 9.4, the relationship between configuration objects and entities, variants and versions can be represented in a three-dimensional space. An entity is composed of a collection of objects at the same revision level. A variant is a different collection of objects at the same revision level and therefore coexists in parallel with other variants. A new version is defined when major changes are made to one or more objects.

A number of different automated approaches to version control have been proposed over the past decade. The primary difference in approaches is the sophistication of the attributes that are used to construct specific versions and variants of a system and the mechanics of the process for construction.

---

3 In this context, the term *entity* refers to all composite objects and basic objects that exist for a baselined SCI. For example, an "input" entity might be constructed with six different software components, each responsible for an input subfunction.

9.5 CHANGE CONTROL

The reality of *change control* in a modern software engineering context has been summed up beautifully by James Bach [BAC98]:

> Change control is vital. But the forces that make it necessary also make it annoying. We worry about change because a tiny perturbation in the code can create a big failure in the product. But it can also fix a big failure or enable wonderful new capabilities. We worry about change because a single rogue developer could sink the project; yet brilliant ideas originate in the minds of those rogues, and a burdensome change control process could effectively discourage them from doing creative work.

Quote:

"The art of progress is to preserve order amid change and to preserve change amid order."

Alfred North Whitehead

Bach recognizes that we face a balancing act. Too much change control and we create problems. Too little, and we create other problems.

For a large software engineering project, uncontrolled change rapidly leads to chaos. For such projects, change control combines human procedures and automated tools to provide a mechanism for the control of change. The change control process is illustrated schematically in Figure 9.5. A *change request*[4] is submitted and evaluated to assess technical merit, potential side effects, overall impact on other configuration objects and system functions, and the projected cost of the change. The results of the evaluation are presented as a *change report,* which is used by a *change control authority* (CCA)—a person or group who makes a final decision on the status and priority of the change. An *engineering change order* (ECO) is generated for each approved change. The ECO describes the change to be made, the constraints that must be respected, and the criteria for review and audit. The object to be changed is "checked out" of the project database, the change is made, and appropriate SQA activities are applied. The object is then "checked in" to the database and appropriate version control mechanisms (Section 9.4) are used to create the next version of the software.

ADVICE

Confusion leads to errors—some of them very serious. Access and synchronization control avoid confusion. Implement them both, even if your approach has to be simplified to accommodate your development culture.

The "check-in" and "check-out" process implements two important elements of change control—access control and synchronization control. *Access control* governs which software engineers have the authority to access and modify a particular configuration object. *Synchronization control* helps to ensure that parallel changes, performed by two different people, don't overwrite one another [HAR89].

Access and synchronization control flow are illustrated schematically in Figure 9.6. Based on an approved change request and ECO, a software engineer *checks out* a configuration object. An access control function ensures that the software engineer has authority to check out the object, and synchronization control *locks* the object in the project database so that no updates can be made to it until the currently checked-out version has been replaced. Note that other copies can be checked-out, but other updates cannot be made. A copy of the baselined object, called the *extracted version,*

4 Although many change requests are submitted during the software support phase, we take a broader view in this discussion. A request for change can occur at any time during the software process.

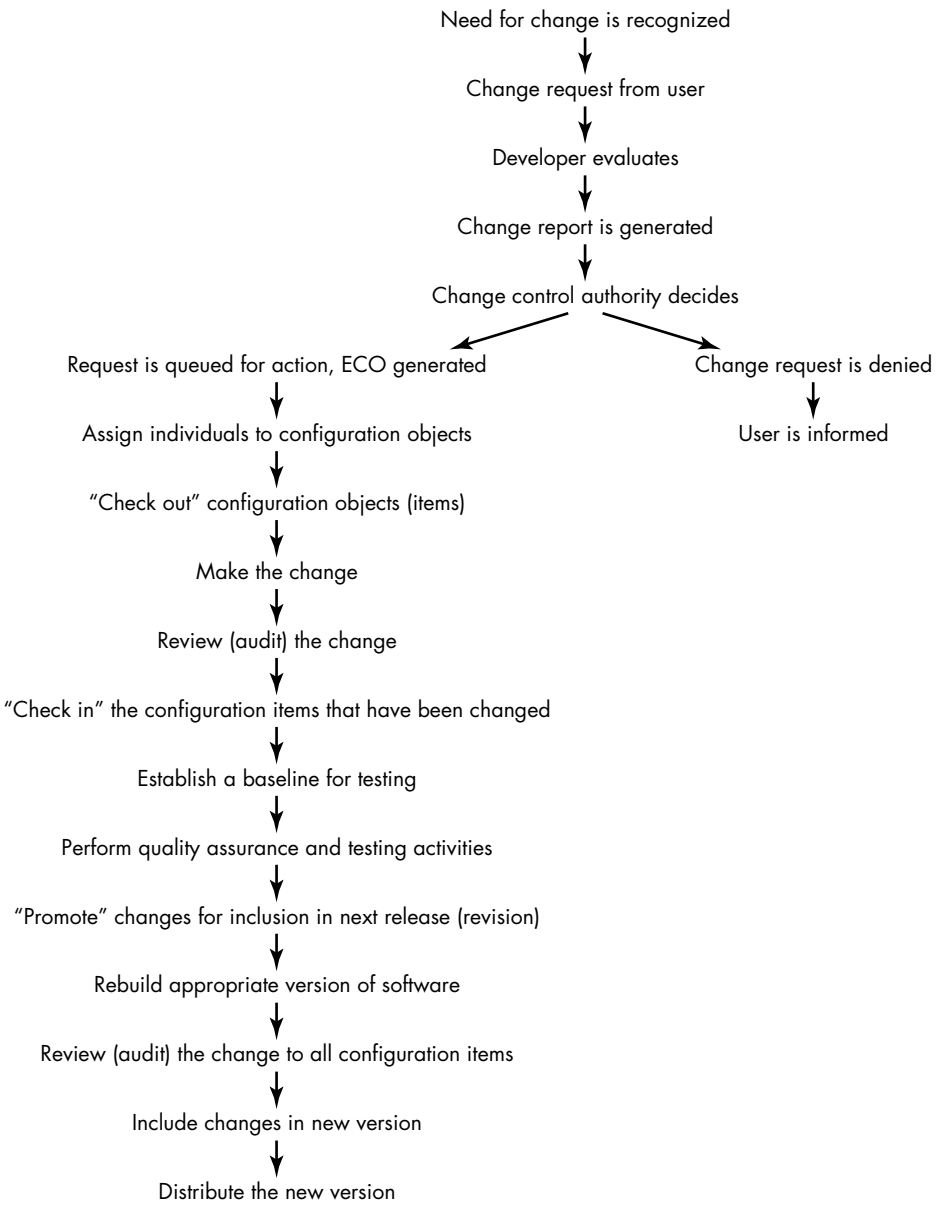
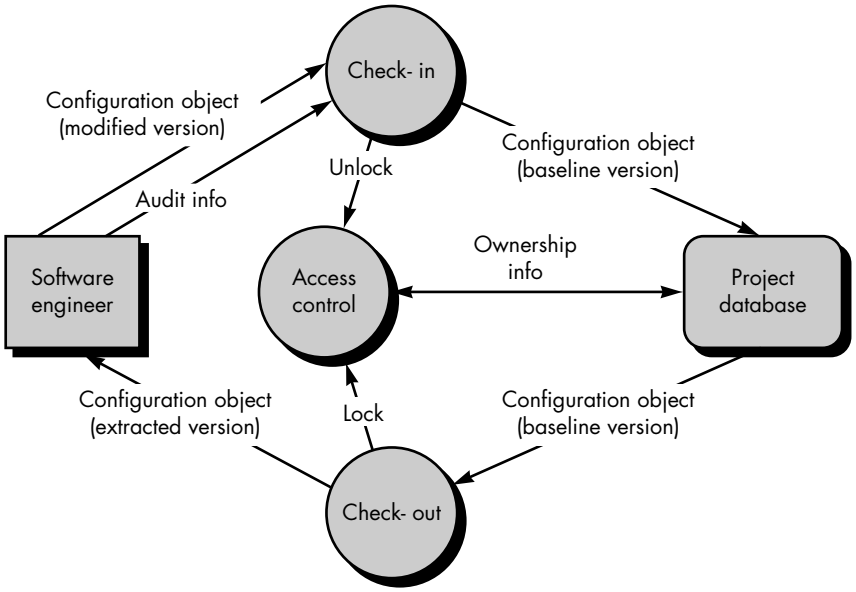


FIGURE 9.5
The change control process

is modified by the software engineer. After appropriate SQA and testing, the modified version of the object is *checked in* and the new baseline object is unlocked.

Some readers may begin to feel uncomfortable with the level of bureaucracy implied by the change control process description. This feeling is not uncommon. Without proper safeguards, change control can retard progress and create unnecessary red tape. Most software developers who have change control mechanisms (unfortunately,

FIGURE 9.6
Access and synchronization control



many have none) have created a number of layers of control to help avoid the problems alluded to here.

Opt for a bit more change control than you think you'll need. It's likely that "too much" will be the right amount.

Prior to an SCI becoming a baseline, only *informal change control* need be applied. The developer of the configuration object (SCI) in question may make whatever changes are justified by project and technical requirements (as long as changes do not affect broader system requirements that lie outside the developer's scope of work). Once the object has undergone formal technical review and has been approved, a baseline is created. Once an SCI becomes a baseline, *project level change control* is implemented. Now, to make a change, the developer must gain approval from the project manager (if the change is "local") or from the CCA if the change affects other SCIs. In some cases, formal generation of change requests, change reports, and ECOs is dispensed with. However, assessment of each change is conducted and all changes are tracked and reviewed.

When the software product is released to customers, *formal change control* is instituted. The formal change control procedure has been outlined in Figure 9.5.

The change control authority plays an active role in the second and third layers of control. Depending on the size and character of a software project, the CCA may be composed of one person—the project manager—or a number of people (e.g., representatives from software, hardware, database engineering, support, marketing). The role of the CCA is to take a global view, that is, to assess the impact of change beyond the SCI in question. How will the change affect hardware? How will the change affect performance? How will the change modify customer's perception of the product? How will the change affect product quality and reliability? These and many other questions are addressed by the CCA.

"Change is inevitable, except from vending machines."

Bumper sticker

9.6 CONFIGURATION AUDIT

Identification, version control, and change control help the software developer to maintain order in what would otherwise be a chaotic and fluid situation. However, even the most successful control mechanisms track a change only until an ECO is generated. How can we ensure that the change has been properly implemented? The answer is twofold: (1) formal technical reviews and (2) the software configuration audit.

The formal technical review (presented in detail in Chapter 8) focuses on the technical correctness of the configuration object that has been modified. The reviewers assess the SCI to determine consistency with other SCIs, omissions, or potential side effects. A formal technical review should be conducted for all but the most trivial changes.

? What are the primary questions that we ask during a configuration audit?

A *software configuration audit* complements the formal technical review by assessing a configuration object for characteristics that are generally not considered during review. The audit asks and answers the following questions:

1. Has the change specified in the ECO been made? Have any additional modifications been incorporated?
2. Has a formal technical review been conducted to assess technical correctness?
3. Has the software process been followed and have software engineering standards been properly applied?
4. Has the change been "highlighted" in the SCI? Have the change date and change author been specified? Do the attributes of the configuration object reflect the change?
5. Have SCM procedures for noting the change, recording it, and reporting it been followed?
6. Have all related SCIs been properly updated?

In some cases, the audit questions are asked as part of a formal technical review. However, when SCM is a formal activity, the SCM audit is conducted separately by the quality assurance group.

9.7 STATUS REPORTING

Configuration status reporting (sometimes called *status accounting*) is an SCM task that answers the following questions: (1) What happened? (2) Who did it? (3) When did it happen? (4) What else will be affected?

The flow of information for configuration status reporting (CSR) is illustrated in Figure 9.5. Each time an SCI is assigned new or updated identification, a CSR entry is made. Each time a change is approved by the CCA (i.e., an ECO is issued), a CSR entry is made. Each time a configuration audit is conducted, the results are reported

as part of the CSR task. Output from CSR may be placed in an on-line database [TAY85], so that software developers or maintainers can access change information by keyword category. In addition, a CSR report is generated on a regular basis and is intended to keep management and practitioners appraised of important changes.

Develop a "need to know list" for every SCI and keep it up-to-date. When a change is made, be sure that everyone on the list is informed.

Configuration status reporting plays a vital role in the success of a large software development project. When many people are involved, it is likely that "the left hand not knowing what the right hand is doing" syndrome will occur. Two developers may attempt to modify the same SCI with different and conflicting intents. A software engineering team may spend months of effort building software to an obsolete hardware specification. The person who would recognize serious side effects for a proposed change is not aware that the change is being made. CSR helps to eliminate these problems by improving communication among all people involved.

9.8 SCM STANDARDS

Over the past two decades a number of software configuration management standards have been proposed. Many early SCM standards, such as MIL-STD-483, DOD-STD-480A and MIL-STD-1521A, focused on software developed for military applications. However, more recent ANSI/IEEE standards, such as ANSI/IEEE Stds. No. 828-1983, No. 1042-1987, and Std. No. 1028-1988 [IEE94], are applicable for non-military software and are recommended for both large and small software engineering organizations.

9.9 SUMMARY

Software configuration management is an umbrella activity that is applied throughout the software process. SCM identifies, controls, audits, and reports modifications that invariably occur while software is being developed and after it has been released to a customer. All information produced as part of software engineering becomes part of a software configuration. The configuration is organized in a manner that enables orderly control of change.

The software configuration is composed of a set of interrelated objects, also called software configuration items, that are produced as a result of some software engineering activity. In addition to documents, programs, and data, the development environment that is used to create software can also be placed under configuration control.

Once a configuration object has been developed and reviewed, it becomes a baseline. Changes to a baselined object result in the creation of a new version of that object. The evolution of a program can be tracked by examining the revision history of all configuration objects. Basic and composite objects form an object pool from which variants and versions are created. Version control is the set of procedures and tools for managing the use of these objects.

Change control is a procedural activity that ensures quality and consistency as changes are made to a configuration object. The change control process begins with a change request, leads to a decision to make or reject the request for change, and culminates with a controlled update of the SCI that is to be changed.

The configuration audit is an SQA activity that helps to ensure that quality is maintained as changes are made. Status reporting provides information about each change to those with a need to know.

REFERENCES

[BAB86] Babich, W.A., *Software Configuration Management,* Addison-Wesley, 1986.
[BAC98] Bach, J., "The Highs and Lows of Change Control," *Computer,* vol. 31, no. 8, August 1998, pp. 113–115.
[BER80] Bersoff, E.H., V.D. Henderson, and S.G. Siegel, *Software Configuration Management,* Prentice-Hall, 1980.
[CHO89] Choi, S.C. and W. Scacchi, "Assuring the Correctness of a Configured Software Description," *Proc. 2nd Intl. Workshop on Software Configuration Management,* ACM, Princeton, NJ, October 1989, pp. 66–75.
[CLE89] Clemm, G.M., "Replacing Version Control with Job Control," *Proc. 2nd Intl. Workshop on Software Configuration Management,* ACM, Princeton, NJ, October 1989, pp. 162–169.
[GUS89] Gustavsson, A., "Maintaining the Evoluation of Software Objects in an Integrated Environment," *Proc. 2nd Intl. Workshop on Software Configuration Management,* ACM, Princeton, NJ, October 1989, pp. 114–117.
[HAR89] Harter, R., "Configuration Management," *HP Professional,* vol. 3, no. 6, June 1989.
[IEE94] *Software Engineering Standards,* 1994 edition, IEEE Computer Society, 1994.
[LIE89] Lie, A. et al., "Change Oriented Versioning in a Software Engineering Database," *Proc. 2nd Intl. Workshop on Software Configuration Management,* ACM, Princeton, NJ, October, 1989, pp. 56–65.
[NAR87] Narayanaswamy, K. and W. Scacchi, "Maintaining Configurations of Evolving Software Systems," *IEEE Trans. Software Engineering,* vol. SE-13, no. 3, March 1987, pp. 324–334.
[REI89] Reichenberger, C., "Orthogonal Version Management," *Proc. 2nd Intl. Workshop on Software Configuration Management,* ACM, Princeton, NJ, October 1989, pp. 137–140.
[TAY85] Taylor, B., "A Database Approach to Configuration Management for Large Projects," *Proc. Conf. Software Maintenance—1985,* IEEE, November 1985, pp. 15–23.
[TIC82] Tichy, W.F., "Design, Implementation and Evaluation of a Revision Control System," *Proc. 6th Intl. Conf. Software Engineering,* IEEE, Tokyo, September 1982, pp. 58–67.

PROBLEMS AND POINTS TO PONDER

9.1. Why is the First Law of System Engineering true? How does it affect our perception of software engineering paradigms.

9.2. Discuss the reasons for baselines in your own words.

9.3. Assume that you're the manager of a small project. What baselines would you define for the project and how would you control them?

9.4. Design a project database system that would enable a software engineer to store, cross reference, trace, update, change, and so forth all important software configuration items. How would the database handle different versions of the same program? Would source code be handled differently than documentation? How will two developers be precluded from making different changes to the same SCI at the same time?

9.5. Do some research on object-oriented databases and write a paper that describes how they can be used in the context of SCM.

9.6. Use an E-R model (Chapter 12) to describe the interrelationships among the SCIs (objects) listed in Section 9.1.2.

9.7. Research an existing SCM tool and describe how it implements control for versions, variants, and configuration objects in general.

9.8. The relations **<part-of>** and **<interrelated>** represent simple relationships between configuration objects. Describe five additional relationships that might be useful in the context of a project database.

9.9. Research an existing SCM tool and describe how it implements the mechanics of version control. Alternatively, read two or three of the papers on SCM and describe the different data structures and referencing mechanisms that are used for version control.

9.10. Using Figure 9.5 as a guide, develop an even more detailed work breakdown for change control. Describe the role of the CCA and suggest formats for the change request, the change report, and the ECO.

9.11. Develop a checklist for use during configuration audits.

9.12. What is the difference between an SCM audit and a formal technical review? Can their function be folded into one review? What are the pros and cons?

FURTHER READINGS AND INFORMATION SOURCES

One of the few books that have been written about SCM in recent years is by Brown, et al. (*AntiPatterns and Patterns in Software Configuration Management,* Wiley, 1999). The authors discuss the things not to do (antipatterns) when implementing an SCM process and then consider their remedies.

Lyon (*Practical CM: Best Configuration Management Practices for the 21st Century,* Raven Publishing, 1999) and Mikkelsen and Pherigo (*Practical Software Configuration Management: The Latenight Developer's Handbook,* Allyn & Bacon, 1997) provide pragmatic tutorials on important SCM practices. Ben-Menachem (*Software Configuration Management Guidebook,* McGraw-Hill, 1994), Vacca (*Implementing a Successful Configuration Change Management Program,* I. S. Management Group, 1993), and Ayer and Patrinnostro (*Software Configuration Management,* McGraw-Hill, 1992) present good overviews for those who need further introduction to the subject. Berlack (*Soft-*

ware Configuration Management, Wiley, 1992) presents a useful survey of SCM concepts, emphasizing the importance of the repository and tools in the management of change. Babich [BAB86] provides an abbreviated, yet effective, treatment of pragmatic issues in software configuration management.

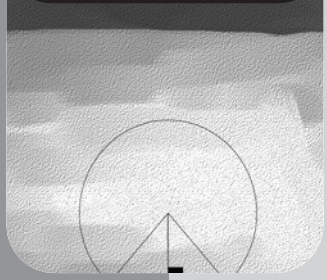
Buckley (*Implementing Configuration Management,* IEEE Computer Society Press, 1993) considers configuration management approaches for all system elements—hardware, software, and firmware—with detailed discussions of major CM activities. Rawlings *(SCM for Network Development Environments,* McGraw-Hill, 1994) is the first SCM book to address the subject with a specific emphasis on software development in a networked environment. Whitgift (*Methods and Tools for Software Configuration Management,* Wiley, 1991) contains reasonable coverage of all important SCM topics, but is distinguished by discussion of repository and CASE environment issues. Arnold and Bohner (*Software Change Impact Analysis,* IEEE Computer Society Press, 1996) have edited an anthology that discusses how to analyze the impact of change within complex software-based systems.

Because SCM identifies and controls software engineering documents, books by Nagle (*Handbook for Preparing Engineering Documents: From Concept to Completion,* IEEE, 1996), Watts (*Engineering Documentation Control Handbook: Configuration Management for Industry,* Noyes Publications, 1993), Ayer and Patrinnostro (*Documenting the Software Process,* McGraw-Hill, 1992) provide a complement to more-focused SCM texts. The March 1999 edition of *Crosstalk* contains a number of useful articles on SCM.

A wide variety of information sources on software configuration management and related subjects is available on the Internet. An up-to-date list of World Wide Web references that are relevant to SCM can be found at the SEPA Web site:
http://www.mhhe.com/engcs/compsci/pressman/resources/scm.mhtml

PART

Three



CONVENTIONAL METHODS FOR SOFTWARE ENGINEERING

In this part of *Software Engineering: A Practitioner's Approach,* we consider the technical concepts, methods, and measurements that are applicable for the analysis, design, and testing of computer software. In the chapters that follow, you'll learn the answers to the following questions:

- How is software defined within the context of a larger system and how does system engineering play a role?
- What basic concepts and principles are applicable to the analysis of software requirements?
- What is structured analysis and how do its various models enable you to understand data, function, and behavior?
- What basic concepts and principles are applied to the software design activity?
- How are design models for data, architecture, interfaces, and components created?
- What basic concepts, principles, and strategies are applicable to software testing?
- How are black-box and white-box testing methods used to design effective test cases?
- What technical metrics are available for assessing the quality of analysis and design models, source code, and test cases?

Once these questions are answered, you'll understand how to build computer software using a disciplined engineering approach.