3 PROJECT MANAGEMENT CONCEPTS

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

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

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

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

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:

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. **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]

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.

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.

3.2.3 The Software Team

There are almost as many human organizational structures for software development as there are organizations that develop software. For better or worse, organizational structure cannot be easily modified. Concern with the practical and political consequences of organizational change are not within the software project manager's scope of responsibility. However, the organization of the people directly involved in a new software project is within the project manager's purview.

The 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.

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:

- 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.

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:

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.

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:

- 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.

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.

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).

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

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.

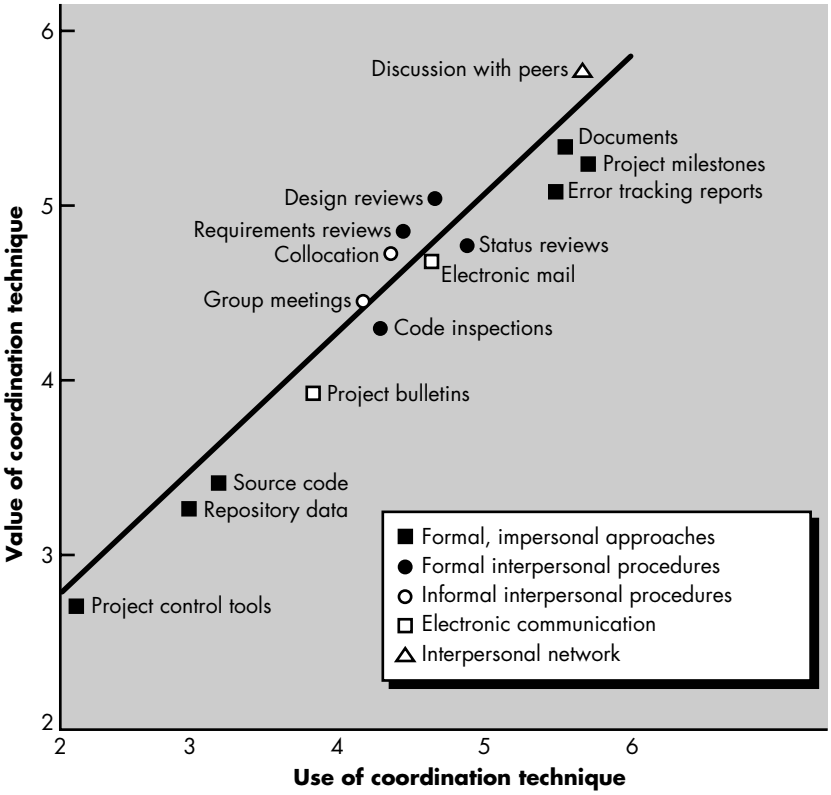
These characteristics of modern software—scale, uncertainty, and interoperability—are facts of life. To deal with them effectively, a software engineering team must establish effective methods for coordinating the people who do the work. To accomplish this, mechanisms for formal and informal communication 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."



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:

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

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.

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

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,

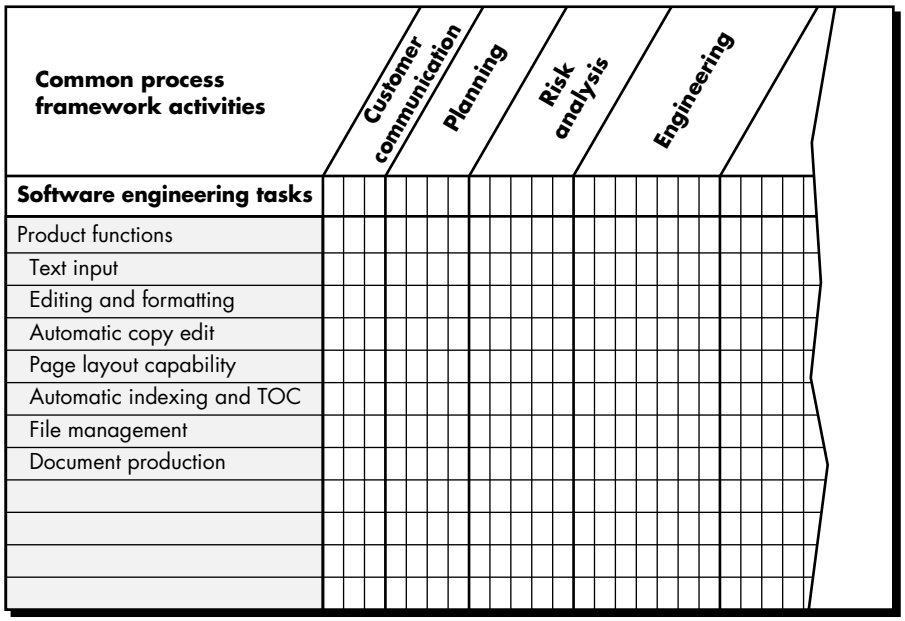
(2) the characteristics of the product itself, and (3) the project environment in which the software team works. When a process model has been selected, the team then defines a preliminary project plan based on the set of common process framework activities. Once the preliminary plan is established, 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):

- *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.

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.



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]

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,

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.

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

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

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:

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.

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]

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.