MANAGING AND LEADING SOFTWARE PROJECTS

RICHARD E. (DICK) FAIRLEY

IEEE computer society

WILEY

A JOHN WILEY & SONS, INC., PUBLICATION

MANAGING AND LEADING SOFTWARE PROJECTS

MANAGING AND LEADING SOFTWARE PROJECTS

RICHARD E. (DICK) FAIRLEY

IEEE computer society

WILEY

A JOHN WILEY & SONS, INC., PUBLICATION

Published by John Wiley & Sons, Inc., Hoboken, New Jersey.
Published simultaneously in Canada.

Library of Congress Cataloging-in-Publication Data is available.

ISBN: 978-0-470-29455-0

Printed in the United States of America.

10 9 8 7 6 5 4 3 2 1

CONTENTS

PREFACE

Too often those who develop and modify software and those who manage software development are like trains traveling different routes to a common destination. The managers want to arrive at the customer's station with an acceptable product, on schedule and within budget. The developers want to deliver to the users a trainload of features and quality attributes; they will delay the time of arrival to do so, if allowed. Sometimes the two trains appear to be on the same schedule, but often one surges ahead only to be sidetracked by traffic of higher priority while the other chugs onward. One or both may be unexpectedly rerouted, making it difficult to rendezvous en route and at the final destination.

Managers traveling on their train often wonder why programmers cannot just write the code that needs to be written, correctly and completely, and deliver it when it is needed. Software developers traveling on their train wonder what their managers do all day. This text provides the insights, methods, tools, and techniques needed to keep both trains moving in unison through their signals and switches and, better yet, shows how they can combine their engines and freight to form a single express train running on a pair of rails, one technical, the other managerial.

By reading this text and working through the exercises, students, software developers, project managers, and prospective managers will learn why

> managing a large computer programming project is like managing any other large undertaking—in more ways than most programmers believe. But in many ways it is different—in more ways than most professional managers expect.[1]

Readers will learn how software projects differ from other kinds of projects (i.e., construction, agricultural, manufacturing, administrative, and traditional engineering projects), and they will learn how the methods and techniques of project management must be modified and adapted for software projects.

[1] *The Mythical Man-Month, Anniversary Edition*, Frederick P. Brooks Jr., Addison Wesley, 1995; pp. x.

Those who are, or will become managers of software projects, will acquire the methods, tools, and techniques needed to effectively manage software projects, both large and small. Software developers, both neophyte student and journeyman/journeywoman professional, will gain an increased understanding of what managers do, or should be doing all day and why managers ask them to do the things they ask/demand. These readers will gain the knowledge they need to become project managers. Those students and software developers who have no desire to become project managers will benefit by gaining an increased understanding of what those other folks do all day and why the seemingly extraneous things they, the developers, are asked to do are important to the success of their projects.

This text is intended as a textbook for upper division undergraduates and graduate students as well as for software practitioners and current and prospective software project managers. Exercises are included in each chapter. Practical hints and guidelines are included throughout the text, thus making it suitable for industrial short courses and for self-study by practitioners and managers.

Chapters 1 through 3 provide the context for the remainder of the text: Chapter 1 provides an introduction to software project management; Chapter 2 covers process models for developing software-intensive systems; Chapter 3 is concerned with establishing the product foundations for software projects.

Chapters 4 through 10 cover the four primary activities of software project management:

- Planning and estimating is covered in Chapters 4 through 6.
- Measuring and controlling is covered in Chapters 7 and 8.
- Managing risk is covered in Chapter 9.
- Leading, motivating, and communicating are covered in Chapter 10.

Chapter 11 covers organizational issues and concludes the text with a summary of 15 guidelines for organizing and leading software engineering teams.

For each topic covered, the approach taken is to present the full scope of activities for the largest and most complex projects and to show how those activities can be tailored, adapted, and scaled to fit the needs of projects of various sizes and complexities.

Learning objectives are presented at the beginning of each chapter and each concludes with a summary of key points from the chapter. Occasional sidebars elaborate the material at hand. An appendix to each chapter relates the topics covered in that chapter to four leading sources of information concerning management of software projects:

1. CMMI-DEV-v1.2 process framework
2. ISO/IEC and IEEE/EIA Standards 12207
3. IEEE/EIA Standard 1058
4. PMI's Body of Knowledge (PMBOK®)

The text is consistent with the guidelines contained in PMBOK and ACM/IEEE curriculum recommendations.

Presentation slides, document templates, and other supporting material for the text and for term projects are available at the following URL:
computer.org/book_extras/fairley_software_projects

Terms used throughout this text are defined in the Glossary at the end of the text. Topics, schedule, and a template for term projects follow the Glossary and included are some hypothetical projects that can be used as the basis for term projects in a course or as examples that practitioners and managers can use to gain experience in preparing software project management plans. Schedule and templates for deliverables for the hypothetic projects are also provided; electronic copies of templates and some software tools are provided at the URL previously cited. Alternatively, practitioners and managers can apply the templates and tools to a past, present, or future project.

A continued example for planning and conducting a project to build the software element of an automated teller system is presented to motivate and explain the material contained in each chapter.

As is well known, one learns best by doing. I strongly recommended that the exercises at the end of each chapter be completed and that progress through the material be accompanied by an extended exercise (i.e., a term project) to develop some elements a project plan for a real or hypothetical software project. The planning exercise can be based on an actual project that the reader has been, is currently, or will be involve in; or it can be based on one of the hypotheticals at the end of the text; or it can be based on a project assigned by the instructor. A week-by-week schedule for completing the term project on a quarter or semester basis is provided. Completion of the planning exercise will result in a report that contains elements similar to those presented in IEEE/EIA Standard 1058 for software project management plans.

The material can be presented in reading/lecture/discussion format or by assigned readings followed by classroom or on-line discussions based on the exercises and the term project.

I am indebted to the pioneers who surveyed the terrain, prepared the roadbed, laid down the tracks, and drove the golden spike so that our project trains can proceed to their destinations. Those pioneers include Fred Brooks, the intellectual father of us all; Winston Royce, who showed us systematic approaches to software development and management of software projects; Barry Boehm, who was the first to address issues of software engineering economics, risk management, and so much more; Tom DeMarco, the master tactician of software development, project management, and peopleware; and the many others who prepared the way for this text. I accept responsibility for any misinterpretations or misstatements of their work. My apologies to those I have failed to credit in the text, either through ignorance or oversight.

Thanks to Mary Jane Fairley, Linda Shafer, and the other reviewers of the manuscript for taking the time to read it and for the many insightful comments they offered. Special thanks to the many students to whom I have presented this material and from whom I have learned as much as they have learned from me.

Teller County, Colorado RICHARD E. (DICK) FAIRLEY

1

INTRODUCTION

In many ways, managing a large computer programming project is like managing any other large undertaking—in more ways than most programmers believe. But in many other ways it is different—in more ways than most professional managers expect.[1]

—Fred Brooks

1.1 INTRODUCTION TO SOFTWARE PROJECT MANAGEMENT

When you become (or perhaps already are) the manager of a software project you will find that experience to be one of the most challenging and most rewarding endeavors of your career. You, as a project manager, will be (or are) responsible for (1) delivering an acceptable product, (2) on the specified delivery date, and (3) within the constraints of the specified budget, resources, and technology. In return you will have, or should have, authority to use the resources available to you in the ways you think best to achieve the project objectives within the constraints of acceptable product, delivery date, and budget, resources, and technology.

Unfortunately, software projects have the (often deserved) reputation of costing more than estimated, taking longer than planned, and delivering less in quantity and quality of product than expected or required. Avoiding this stereotypical situation is the challenge of managing and leading software projects.

There are four fundamental activities that you must accomplish if you are to be a successful project manager:

[1] *The Mythical Man-Month, Anniversary Edition*, Frederick P. Brooks Jr., Addison Wesley, 1995; p. x.

1. planning and estimating,
2. measuring and controlling,
3. communicating, coordinating, and leading, and
4. managing risk.

These are the major themes of this text.

1.2 OBJECTIVES OF THIS CHAPTER

After reading this chapter and completing the exercises, you should understand:

- why managing and leading software projects is difficult,
- the nature of project constraints,
- a workflow model for software projects,
- the work products of software projects,
- the organizational context of software projects,
- organizing a software development team,
- maintaining the project vision and product goals, and
- the nature of process frameworks, software engineering standards, and process guidelines.

Appendix 1A to this chapter provides an introduction to elements of the following frameworks, standards, and guidelines that are concerned with managing software projects: the SEI Capability Maturity Model® Integration CMMI-DEV-v1.2, ISO/IEC and IEEE/EIA Standards 12207, IEEE/EIA Standard 1058, and the Project Management Body of Knowledge (PMBOK®). Terms used in this chapter and throughout this text are defined in a glossary at the end of the text. Presentation slides for this chapter and other supporting material are available at the URL listed in the Preface.

1.3 WHY MANAGING AND LEADING SOFTWARE PROJECTS IS DIFFICULT

A *project* is a group of coordinated activities conducted within a specific time frame for the purpose of achieving specified objectives. Some projects are personal in nature, for example, building a dog house or painting a bedroom. Other projects are conducted by organizations. The focus of this text is on projects conducted within software organizations. In a general sense, all organizational projects are similar:

- objectives must be specified,
- a schedule of activities must be planned,
- resources allocated,
- responsibilities assigned,

- work activities coordinated,
- progress monitored,
- communication maintained,
- risk factors identified and confronted, and
- corrective actions applied as necessary.

In a specific sense, the methods, tools, and techniques used to manage a project depend on the nature of the work to be accomplished and the work products to be produced. Manufacturing projects are different from construction projects, which are different from agricultural projects, which are different from computer hardware projects, which are different from software engineering projects, and so on. Each kind of project, including software projects, adapts and tailors the general procedures of project management to accommodate the unique aspects of the development processes and the nature of the product to be developed.

Fred Brooks has famously observed that four essential properties of software differentiate it from other kinds of engineering artifacts and make software projects difficult[2]:

1. complexity,
2. conformity,
3. changeability, and
4. invisibility of software.

1.3.1 Software Complexity

Software is more complex, for the effort and the expense required to construct it, than most artifacts produced by human endeavor. Assuming it costs $50 (USD) per line of code to construct a one-million line program (specify, design, implement, verify, validate, and deliver it), the resulting cost will be $50,000,000. While this is a large sum of money, it is a small fraction of the cost of constructing a complex spacecraft, a skyscraper, or a naval aircraft carrier.

Brooks says, "Software entities are more complex for their *size* [emphasis added] than perhaps any other human construct, because no two parts are alike (at least above the statement level)." [3] It is difficult to visualize the size of a software program because software has no physical attributes; however, if one were to print a one-million line program the stack of paper would be about 10 feet (roughly 3 meters) high if the program were printed 50 lines per page. The printout would occupy a volume of about 6.5 cubic feet. Biological entities such as human beings are of similar volume and they are far more complex than computer software, but there are few, if any, human-made artifacts of comparable size that are as complex as software.

The complexity of software arises from the large number of unique, interacting parts in a software system. The parts are unique because, for the most part, they are encapsulated as functions, subroutines, or objects and invoked as needed rather

[2] *Ibid*, pp. 182–186.
[3] *Ibid*, p. 182.

than being replicated. Software parts have several different kinds of interactions, including serial and concurrent invocations, state transitions, data couplings, and interfaces to databases and external systems. Depiction of a software entity often requires several different representations to portray the numerous static structures, dynamic couplings, and modes of interaction that exist in computer software.

A seemingly "small" change in requirements is one of the many ways that complexity of the product may affect management of a project. Complexity within the parts and in the connections among parts may result in a large amount of evolutionary rework for the "small" change in requirements, thus upsetting the ability to make progress according to plan. For this reason many experienced project managers say there are no small requirements changes. Size and complexity can also hide defects that may not be discovered immediately and thus require additional, unplanned corrective rework later.

1.3.2 Software Conformity

Conformity is the second issue cited by Brooks. Software must conform to exacting specifications in the representation of each part, in the interfaces to other internal parts, and in the connections to the environment in which it operates. A missing semicolon or other syntactic error can be detected by a compiler but a defect in the program logic, or a timing error caused by failure to conform to the requirements may be difficult to detect until encountered in operation. Unlike software, tolerance among the interfaces of physical entities is the foundation of manufacturing and construction; no two physical parts that are joined together have, or are required to have, exact matches. Eli Whitney (of cotton gin fame) realized in 1798 that if musket parts were manufactured to specified tolerances, interchangeability of similar (but not identical) parts could be achieved.

There are no corresponding tolerances in the interfaces among software entities or between software entities and their environments. Interfaces among software parts must agree exactly in numbers and types of parameters and kind of couplings. There are no interface specifications for software stating that a parameter can be "an integer plus or minus 2%."

Lack of conformity can cause problems when an existing software component cannot be reused as planned because it does not conform to the needs of the product under development. Lack of conformity might not be discovered until late in a project, thus necessitating development and integration of an acceptable component to replace the one that cannot be reused. This requires unplanned allocation of resources and can delay product completion. Complexity may have made it difficult to determine that the reuse component lacked the necessary conformity until the components it would interact with were completed.

1.3.3 Software Changeability

Changeability is Brooks's third factor that makes software projects difficult. Software coordinates the operation of physical components and provides the functionality in software-intensive systems.[4] Because software is the most easily changed element (i.e., the most malleable) in a software-intensive system, it is the most frequently changed element, particularly in the late stages of a project. Changes may occur because customers change their minds; competing products change; mission objectives change; laws, regulations, and business practices change; underlying hardware and software technology changes (processors, operating systems, application packages); and/or the operating environment of the software changes. If an early version of the final product is installed in the operating environment, it will change that environment and result in new requirements that will require changes to the product. Simply stated, now that the new system enables me to do A and B, I would like for it to also allow me to do C, or to do C instead of B.

Each proposed change in product requirements must be accompanied by an analysis of the impact of the change on project work activities:

- what work products will have to be changed?
- how much time and effort will be required?
- who is available to make the changes?
- how will the change affect your plans for schedule, budget, resources, technology, other product features, and the quality attributes of the product?

The goal of impact analysis is to determine whether a proposed change is "in scope" or "out of scope." In-scope changes to a software product are changes that can be accomplished with little or no disruption to planned work activities. Acceptance of an out-of-scope change to the product requirements must be accompanied by corresponding adjustments to the budget, resources, and/or schedule; and/or modification or elimination of other product requirements. These actions can bring a proposed out-of-scope requirement change into revised scope.

A commonly occurring source of problems in managing software projects is an out-of-scope product change that is not accompanied by corresponding changes to the schedule, resources, budget, and/or technology. The problems thus created include burn-out of personnel from excessive overtime, and reduction in quality because tired people make more mistakes. In addition reviews, testing, and other quality control techniques are often reduced or eliminated because of inadequate time and resources to accomplish the change and maintain these other activities.

1.3.4 Software Invisibility

The fourth of Brooks's factors is invisibility. Software is said to be invisible because it has no physical properties. While the effects of executing software on a digital computer are observable, software itself cannot be seen, tasted, smelled, touched, or heard. Our five human senses are incapable of directly sensing software; software is thus an intangible entity. Work products such as requirements specifications, design documents, source code, and object code are representations of software, but

[4] Software-intensive systems contain one or more digital devices and may include other kinds of hardware plus trained operators who perform manual functions. Nuclear reactors, modern aircraft, automobiles, network servers, and laptop computers are examples of software-intensive systems.

they are not the software. At the most elemental level, software resides in the magnetization and current flow in an enormous number of electronic elements within a digital device. Because software has no physical presence we use different representations, at different levels of abstraction, in an attempt to visualize the inherently invisible entity.

Because software cannot be directly observed as can, for example, a building under construction or an agricultural plot being prepared for planting, the techniques presented in this text can be used to determine the true state of progress of a software project. An unfortunate result of failing to use these techniques is that software products under development are often reported to be "almost complete" for long periods of time with no objective evidence to support or refute the claim; this is the well-known "90% complete syndrome" of software projects. Many software projects have been canceled after large investments of effort, time, and money because no one could objectively determine the status of the work products or provide a credible estimate of a completion date or the cost to complete the project. Sad but true, this will occur again. You do not want to be the manager of one of those projects.

1.3.5 Team-Oriented, Intellect-Intensive Work

In addition to the essential properties of software (complexity, conformity, changeability, and invisibility), one additional factor distinguishes software projects from other kinds of projects: *software projects are team-oriented, intellect-intensive endeavors*. In contrast, assembly-line manufacturing, construction of buildings and roads, planting of rice, and harvesting of fruit are labor-intensive activities; the work is arranged so that each person can perform a task with a high degree of autonomy and a small amount of interaction with others. Productivity increases linearly with the number of workers added; the work will proceed roughly twice as fast if the number of workers is doubled. Although labor-saving machines have increased productivity in some of these areas, the roles played by humans in these kinds of projects are predominantly labor-intensive.

Software is developed by teams of individuals who engage in creative problem solving. Teams are necessary because it would take too much time for one person to develop a modern software system and because it is unlikely that one individual would possess the necessary range of skills. Suppose, for example, that the total effort to develop a software product or system[5] results in a productivity level of 1000 lines of code per staff-month (more on this later). A one million line program would require 1000 staff-months. Because effort (staff-months) is the product of people and time, it would require 1 person 1000 months (about 83 years) to complete the project.

A feasible combination of people and time for a 1000 staff-month project might be a team of 50 people working for 20 months but not 1000 people working for 1 month or even 200 people working for 5 months. The later proposals (1000 × 1 and

[5] Software *products* are built by *vendors* for sale to numerous customers; software *systems* are built by *contractors* for specific customers on a contractual basis. The terms "system" and "product" are used interchangeably in this text unless the distinction is important; the distinction will be clarified in these cases.

200×5) are not feasible because scheduling constraints among work activities dictate that some activities cannot begin before other work activities are completed: you can't design (some part of a system) without some corresponding requirements, you should not write code without a design specification for (that part of) the system, you cannot review or test code until some code has been written, you cannot integrate software modules until they are available for integration, and so on.

Adding people to a software development team does not, as a rule, increase overall productivity in a linear manner because the increased overhead of communicating with and coordinating work activities among the added people decreases the productivity of the existing team. To cite Fred Brooks once again, the number of communication paths among n workers is $n(n-1)/2$, which is the number of links in a fully connected graph. Five workers have 20 communication paths, 10 have 45 paths, and 20 have 190. Increasing the size of a programming team from 5 to 10 members might, for example, might increase the production rate of the team from 5000 lines of code per week to 7500 lines of code per week, but not 10,000 lines of code per week as would occur with linear scaling. In *The Mythical Man-Month*, Brooks described this phenomenon as Brooks's law[6]:

Adding manpower to a late software project makes it later.

Brooks's law is based on three factors:

1. the time required for existing team members to indoctrinate new team members,
2. the learning curve for the new members, and
3. the increased communication overhead that results from the new and existing members working together.

Brooks's law would not be true if the work assigned to the new members did not invoke any of these three conditions.

A simile that illustrates the issues of team-oriented software development is that of a team of authors writing a book as a collaborative project; a team of authors is very much like a team of software developers. In the beginning, requirements analysis must be performed to determine the kind of book to be written and the constraints that apply to writing it. The number and skills of team members will constrain the kind and size of book that can be written by the available team of authors within a specified time frame. Constraints may include the number of people on the writing team, knowledge and skills of team members, the required completion date, and the word-processing hardware and software available to be used.

Next the structure of the book must be designed: the number of chapters, a brief synopsis of each, and the relationships (interfaces) among chapters must be specified. The book may be structured into sections that contain several chapters each (subsystems), or the text may be structured into multiple volumes (a system of systems). The dynamic structure of the text may flow linearly in time or it may move backward and forward in time between successive chapters; primary and

[6] *Ibid*, pp. 25 and 274.

secondary plot lines may be interleaved. An important constraint is to develop a design structure that will allow each team member to accomplish some work while other team members are accomplishing their work so that the work activities can proceed in parallel. Some books are cleverly structured to have multiple endings; readers choose the one they like.

Design details to be decided include the format of textual layout, fonts to be used, footnoting and referencing conventions, and stylistic guidelines (use of active and passive voice, use of dialects and idioms). Writing of the text occurs within a predetermined schedule of production that includes reviews by other team members (peer reviews) and independent reviews by copy editors (independent verification). Revisions determined by the reviews must be accomplished. The goal of the writing team is to produce a seamless text that appears to have been written by one person in a single setting.

A deviation from the planned narrative by one or more team members might produce a ripple effect that would require extensive revision of the text. If the completed book were software, a single punctuation or grammatical error in the text would render the book unreadable until the writers or their copy editor repaired the defect. An editor determines that each iteration of elements of the text satisfy the conditions placed on it by other elements (verification). Finally, reviews by critics and purchases by readers will determine the degree to which the book satisfies its intended purpose in its intended environment (validation).

The various development phases of writing (analysis, high-level design, detailed design, implementation, peer review, independent verification, revision, and validation) are creative activities and thus rarely occur in linear, sequential fashion. Conducting analysis, preparing and revising the design of the text, and production, review, and revision of the various parts may be overlapped, interleaved, and iterated. Team members must each do their assigned tasks, coordinate their work with other team members, and communicate ideas, problems, and changes on a continuous basis. The narrative above depicts a so-called Plan-driven approach to writing a book and, by analogy, to developing software. An alternative is to pursue an Agile approach by which the team members start with a basic concept and evolve the text in an iterative manner. This approach can be successful:

- if the team is small, say five or six members (to limit the complexity of communication);
- if all members have in mind a common understanding of the desired structure of the text (i.e., a "design metaphor");
- if there is a strict page limit and a completion date (the project constraints);
- if each iteration occurs in one or a few days (to facilitate ongoing revisions in structure; known as "refactoring"); and
- if a knowledgeable reader (known as the "customer") is available to review each iteration and provide guidance for the contents of the next iteration.

In some cases, the team members may work in pairs ("pair programming") to enhance synergy of effort.

In reality, most software projects incorporate elements of a plan-driven approach and an agile approach. When pursuing an agile approach, the team members must understand the nature of the desired product to be delivered, a design metaphor must be established, and the constraints on schedule, budget, resources, and technology that must be observed; thus some requirements definition, design, and project planning must be done. Those who pursue a plan-driven strategy often pursue an iterative (agile) approach to developing, verifying, and validating the product to be delivered; frequent demonstrations provide tangible evidence of progress and permit incorporation of changes in an incremental manner.

The approach taken in this text is to present a plan-driven strategy, based on iterative development, that is suitable for the largest and most complex projects, and to show how the techniques can be tailored and adapted to suit the needs of small, simple projects as well as large, complex ones. Process models for software development are presented in Chapter 2.

Over time humans have learned to conduct agricultural, construction, and manufacturing projects that employ teams of workers who accomplish their tasks efficiently and effectively.[7] Because software is characterized by complexity, conformity, changeability, and invisibility, and because software projects are conducted by teams of individuals engaged in intellect-intensive teamwork, we humans are not always as adept at conducting software projects as we are at conducting traditional kinds of projects in agriculture, construction, and manufacturing. Nevertheless, the techniques presented in this text will help you manage software projects efficiently and effectively, that is, with economical use of time and resources to achieve desired outcomes.

Your role as project manager is to plan and coordinate the work activities of your project team so that the team can accomplish more working in a coordinated manner than could be accomplished by each individual working with total autonomy.

1.4 THE NATURE OF PROJECT CONSTRAINTS

Many of the problems you will encounter, or have encountered, in software projects are caused by difficulties of management and leadership (i.e., planning, estimating, measuring, controlling, communicating, coordinating, and managing risk) rather than technical issues (i.e., analysis, design, coding, and testing). These difficulties arise from multiple sources; some you can control as a project manager and some you can't. Factors you can't control are called *constraints*, which are limitations imposed by external agents on some or all of the operational domain, operational requirements, product requirements, project scope, budget, resources, completion date, and platform technology. Table 1.1 lists some typical constraints for software projects and provides brief explanations.

The *operational domain* is the environment in which the delivered software will be used. Operational domains include virtually every area of modern society, including health care, finance, transportation, communication, entertainment, business, and manufacturing environments. Understanding the operational domain in which the software will operate is essential to success. *Operational requirements* describe the

[7] To be *efficient* is to accomplish a task without wasting time or resources; to be *effective* is to obtain the desired result.

TABLE 1.1 Typical constraints on software projects

Constraint	Explanation
Operational domain	Environment of the users
Operational requirements	Users' needs and desires
Product requirements	Functional capabilities and quality attributes
Scientific knowledge	Algorithms and data structures
Process standards	Ways of conducting work activities
Project scope	Work activities to be accomplished
Resources	Assets available to conduct a project
Budget	Money used to acquire resources
Completion date	Delivery date for work products
Platform technology	Software tools and hardware/software base
Business goals	Profit, stability, growth
Ethical considerations	Serving best interests of humans and society

users' view (i.e., the external view) of the system to be delivered. Some desired features, as specified in the operational requirements, may be beyond the current state of scientific knowledge, either at large or within your organization. *Product requirements* are the developers' view (i.e., the internal view) of the system to be built; they include the functional capabilities and quality attributes the delivered product must possess in order to satisfy the operational requirements.

Process standards specify ways of conducting the work activities of software projects. Your organization may have standardized ways of conducting specific activities, such as planning and estimating projects, and measuring project factors such as conformance to the schedule, expenditure of resources, and measurement of quality attributes of the evolving product. In some cases the customer may specify standards and guidelines for conducting a project. Four of the most commonly used frameworks for process standards are the Capability Maturity Model Integration (CMMI), ISO/IEEE Standard 12207, IEEE Standard 1058, and the Project Management Body of Knowledge (PMBOK). Elements of these standards and guidelines are contained in appendixes to the chapters of this text.

The *scope* of a project is the set of activities that must be accomplished to deliver an acceptable product on schedule and within budget. *Resources* are the assets, both corporate and external, that can be applied to the project. Resources have both quality and quantity attributes; for example, you may have a sufficient number of software developers available (quantity of assets), but they may not have the necessary skills (quality of assets). The *budget* is the money available to acquire and use resources; the budget for your project may be constrained so that resources available within the organization cannot be utilized. The *completion date* is the day on which the product must be finished and ready for delivery. In some cases there may be multiple completion dates on which subsets of the final product must be delivered. The constrained delivery date(s) may be unrealistic.

Platform technology includes the set of methods, tools, and development environments used to produce or modify a software product. Examples include tools to develop and document requirements and designs, compilers and debuggers to gen-