Chapter 1

Introduction

> Under any social order from now to Utopia a management is indispensable and all enduring . . . the question is not "Will there be a management elite?" but "What sort of elite will it be?"
>
> —Sidney Webb

Overview

The five papers selected for this introductory chapter provide insights into the concerns and challenges associated with software management and the framework composed to study ways to deal with them. Although each paper makes its own unique point, the majority of the papers were selected because they examine the root causes of what many of us in the industry have called "the software crisis." All too often, managers tend to treat the symptoms and not the root causes of their problems. They then use "gut feel" instead of "hard" data to develop a "quick fix" to the dilemma. For example, managers might try to tie task progress to an impossible schedule when they are falling behind instead of trying to convince their bosses that rescheduling based upon more realistic goals is a better option. They may not realize that rescheduling does not mean the end date has to change. In addition to permitting them to dedicate more time and effort to the tasks that are causing the delays, rescheduling allows them to refocus their resources and take advantage of opportunities to perform other tasks that are not on the critical path in parallel.

The papers included in this chapter were selected to provide you with help in figuring out the root causes of your problems. In the example in the previous paragraph, should progress be considered to be falling behind schedule because the people are not productive enough, the task is tougher to perform than expected, or the original schedule is overly optimistic? These are difficult questions. To develop answers, I have selected papers that establish frameworks to help you understand how different software process, product and people considerations relate to each other and the many life cycle management options that are available for use today. These frameworks are important because they establish a foundation that we will build upon and reference in later chapters in this tutorial.

Article Summaries

Software Management's Seven Deadly Sins, by Donald Reifer. This article is a reprint of an editorial I wrote for *IEEE Software* magazine about my reflections on the state of software and software management worldwide. The paper summarizes the results of a straw poll of experts from 20 organizations who identified what they believed were the major problems relative to managing software. What is surprising about the results is that they reflect problems that have been haunting software managers for the past three decades. The seven sins include: volatile requirements, poor planning, unrealistic schedules and budgets, inadequate controls, undercapitalization, the "we're different" syndrome, and lack of focus on quality. Read on if you are interested in reflections on what we have learned about these problems during the past 30 years.

Principles of Software Engineering Project Management, by Donald Reifer. I originally wrote this article to serve as an introduction to this tutorial. The article reinforces my message that software can be managed using classical project-management approaches. The article ties chapters together and provides you with a road map through this tutorial. The paper has been updated for this volume to reflect the state of the practice of software management and the current content of this volume. The paper's findings, conclusions, and recommendations are presented as principles that you can use to guide your management actions. It is important to note that these principles have not changed during the past decade. They provide you with basic truths or rules for managing software people, projects, and organizations.

The "3P's" of Software Management, by Donald Reifer. This paper communicates principles devised to improve how we address what I call the "3P's" of software management (i.e., the processes we use, the products we generate, and the people who perform the work on our software projects). The paper has been updated to show more clearly how these three important

factors are related to one another and the methods, tools, and techniques that we use for software project management. My hypothesis in the paper is that you can increase your chances of success on a software project by reducing the conflicts that arise between the "3P's." Such conflicts arise as the factors take on different levels of importance as the project unfolds.

Why Big Software Projects Fail: The 12 Key Questions, by Watts Humphrey. Answers to the twelve questions asked within this article reveal that managers of large software projects can improve their ability to succeed by looking at past failures and learning from them. The answers to the questions posed provide the reader with insights provided by one of the old hands in software management. This piece is truly thought provoking and stimulating, especially for those professionals who are embarking on the task of managing software projects, both large and small. I recommend that you ponder its message and use the results accordingly.

Critical Success Factor in Software Projects, by John Reel. This excellent article looks at the things you can do on a software project to increase its chances of success. It suggests that you start off on the right foot by setting realistic objectives and expectations for those working on the project. Then, it recommends that you focus on building the team by carefully selecting the members and furnishing them with what they need to get the job done. To maintain your momentum, the paper recommends that you focus on tracking progress, keeping attrition low, and gathering the "hard" data to make smart decisions. The paper concludes by suggesting that you put a postmortem process in place, allowing you to learn from your mistakes.

Key Terms

For those worried about terminology, I have included an updated and expanded glossary at the end of this tutorial volume. To communicate the meanings of the words, I have sometimes taken liberties with standard definitions to make them more understandable. Common acronyms used in this chapter follow in a paragraph that follows right after the definitions.

The fourteen terms, which are defined as follows, are important to understanding the articles provided in this introduction:

1. **Activity.** A major unit of work to be completed in achieving the goals of a software project. An activity has precise starting and ending dates, has a set of tasks that need to be done, consumes resources, and results in work products. An activity may contain other activities arranged in a hierarchical order.
2. **Budget.** In management, a statement of expected results expressed in numerical terms.
3. **Case study.** An example employed to communicate lessons learned from trial use of a concept or idea.
4. **Critical success factors.** Those characteristics, conditions, or variables that have a direct influence on your customer's satisfaction with the products and services that you offer.
5. **Lessons learned.** The knowledge or experience gained by actually completing the project.
6. **Milestone.** A scheduled event for which some person is held accountable and that is used to demonstrate progress.
7. **Project.** An organized undertaking that uses human and physical resources, done once, to accomplish a specific goal.
8. **Project management.** The system of management established to focus resources on achieving project goals. Project management has been defined as the art of creating the illusion that any outcome is the result of a series of predetermined, deliberate acts when, in fact, it is not.
9. **Practice.** In management, a preferred course of action for getting the job done.
10. **Resources.** In management, the time, staff, capital, and money made available to a project to perform a service or build a project.
11. **Schedule.** The actual calendar time budgeted for accomplishing the goals established for activities or tasks at hand.
12. **Slack.** In networks, the term used to refer to marginal time available to complete a task or activity.
13. **Task.** In management, the smallest unit of work subject to management accountability.
14. **Tracking.** In management, the process of identifying cost and schedule variances by comparing actual expenditures to projects.

Common Acronyms

The following acronyms are used within the articles in this chapter:

3P's	Process, Product and People
CMM	Capability Maturity Model

CMMI	Capability Maturity Model Integration
GUI	Graphical User Interface
IEEE	Institute of Electrical and Electronics Engineers
IS	Information Systems
PC	Personal Computer
PBS	Product Breakdown Structure
PERT	Program Evaluation and Review Technique
PMBOK™	Project Management Body of Knowledge
SEI	Software Engineering Institute
WBS	Work Breakdown Structure

For Your Bookshelf

Although there are many books and articles on management theory, few of them apply this theory within the context of software projects. In addition, few of these publications provide software project managers with help in identifying issues, bounding solutions, and resolving their day-to-day problems. The texts that I have listed in the Annotated Bibliography appearing at the back of this tutorial volume bridge the gap and provide practical guidance aimed at helping to get the software management job done. They do this by providing readers with examples that they can relate to.

Many of the books on this list are relatively new. I have deliberately tried to make sure that these additions cover the latest concepts. I have eliminated most of the more technically oriented texts and referred readers instead to more management-oriented books like McConnel's *Software Project Survival Guide,* Wiegers's classic *Creating a Software Engineering Culture,* and my book, *Making the Software Business Case: Improvement by the Numbers.* I have also kept several classics on the list of recommended readings, as they are easy to read and contain important messages. For example, I believe that everyone in the field should read the *The Mythical Man-Month* by Fred Brooks. His messages are as relevant today as they were when he first published the book almost 40 years ago.

In addition, the *Project Management Body of Knowledge* (PMBOK™), prepared by the Project Management Institute, provides an excellent framework for putting the lessons learned and discussed into action on your project. I highly recommend that you become familiar with this work.

I will point out books you should consider for your bookshelf in my introduction to each chapter of this volume. I will also provide you with pointers to additional information that is available to assist you in these sections.

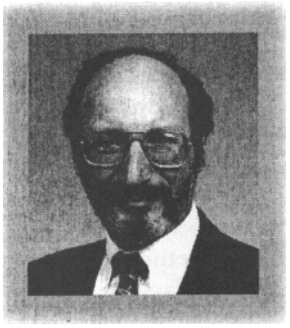
Finally, I will put updates to this volume and errata on my Web site (go to www. reifer. corn). Use of the site will allow me to keep you posted on developments in the software management field.

Software Management's Seven Deadly Sins

Donald J. Reifer

During the past month, I have participated in an online discussion to look at the state of software and software management worldwide. What a revelation! Apparently, at least for the past several years, I have been living a sheltered existence.

That's because most of my consulting clients have had their acts together when it came to software management. These software organizations run like businesses, delivering what they promise—on schedule and within budget. Like well-lubricated machines, they crank out products for their customers—in the aerospace, process control, medical, petrochemical, and telecommunications industries—that work the first time around. These firms have problems, as you might expect, but not overwhelming ones. Most track progress, use metrics, and employ risk management techniques to solve their problems early.



Others in the discussion group contended that I haven't been living in the real world. They argued that software organizations still have a tainted reputation because most software groups fail to deliver what they promise when they promise it. In many firms, senior managers still view software with disdain. To them, it represents the tall pole in the tent: with hardware now typically purchased off the shelf, software consumes most of their engineering resources and represents most of the risk their firms face. As evidence, my interlocutors pointed to statistics recently published by the Software Engineering Institute (SEI), which indicate that most firms recently rated using the Software Capability Maturity Model are still at Level 1.[1] Group members also pointed to the horror stories appearing far too often in the professional literature, documenting what many call death-march projects and software mismanagement.

I decided to poll my industry friends to determine who was right. I drafted questions and conducted fact-finding using email. In all, senior managers from 20 organizations responded to my questions. To my surprise, most of them indicated that we still have major problems when it comes to managing software (see Table 1). Even more surprisingly, this table could have been built 20 years ago. Have we made *no* progress?

While these findings are not statistically significant, they are revealing. Most of these senior managers view software as a drain on their organizations rather than a contributor. Many think software people are unmanageable prima donnas. Many perceive software organizations as unable to deliver what they promise when they promise it. Interestingly enough, when I called several of these managers and asked them to characterize their firms' software groups, three themes echoed through their answers: software groups never seem to have the time to do the job right, their cost and schedule expectations are unrealistic, and I'm scared of them (because these managers do not understand software).

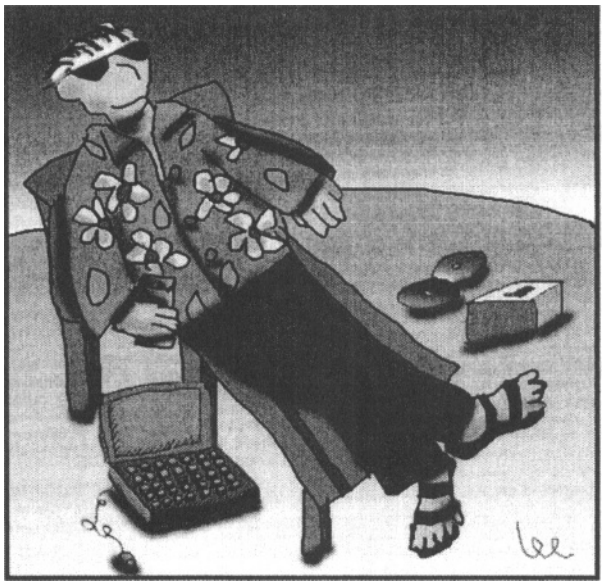
As one of the field's old-timers, I remember how things were the 1970s and early 1980s. Then, as companies increasingly

Reprinted from *IEEE Software*, March/April, 20001.

bought their hardware off the shelf, they discovered that their products were becoming software-intense. Observers likened software development to hardware engineering, and the "software crisis" was on everyone's lips. Of course, the public did not have a clue about software. The PC was new and general use of the Internet was far in the future. To cope with this so-called software crisis, leading firms pursued software initiatives. Most embraced the following three-pronged attack:

- Standardize the process.
- Standardize the product.
- Professionalize the workforce.

The leaders among them also educated their senior members to set expectations. In the late 1980s, the SEI software maturity model came out.[2] This model formed the framework that enabled many firms to adopt the practices, methods, and tools needed to pursue process improvement. In the mid-1990s, architecture technology became the rage.[3] These same firms then could use their processes to develop standard products using product-line management techniques.[4] Because the technology used to develop software continued to change throughout this period, focus shifted to developing the workforce's skills. However, entire new industries emerged and software development moved away from the world of the large monoliths to the Web.



Sally Lee

As Table 1 illustrates, some of us are apparently reliving this history. If so, perhaps we can reorient the solutions that worked in the past to help now. Let's turn our attention to what I call the seven sins of software management[5] and see how solutions that worked in the past might again be our salvation.

Sin 1: Volatile requirements

In the old days, requirements were where the action was. The software industry developed better specification techniques to reduce volatility and get the user involved in the process early. We adopted object-oriented techniques to provide better mapping from problem to solution domains. We learned that we couldn't stabilize the software architecture and design if we let our customers alter requirements at will.

Today's quick-to-market projects seem to be teaching us this lesson anew. Although managers of such projects might use enlightened techniques such as prototyping and modern paradigms such as Mbase[6] and the Rational Unified Process,[7] they succeed only marginally when their organizations let requirements change whenever marketing staffs or customers and users suggest new features and functions. Clearly, better requirements-management processes are fundamental to making improvements.

Sin 2: Poor planning

Years ago, project teams would forsake requirements development and planning because management wanted them to get on with the coding. To cope with this foolishness, we likened software to hardware to illustrate the need for improvement. Firms developed planning templates only after they developed or bought into a life-cycle methodology. Then they brought in classical project-planning tools and techniques such as work breakdown structures and milestone schedules that worked for other disciplines. The software development community learned that we had to plan our future work in detail so we

Table 1

Software Management Straw Poll Findings

Question	Yes (%)	No (%)
Are your software organizations perceived as well managed?	18	82
Do these software organizations deliver what they promise when they promise it?	34	66
Do your customers and users view the products you deliver as high quality?	25	75
Do you employ classical management tools and techniques to manage software deliveries?	55	45
Does your senior management view software organizations as contributing to the bottom line?	30	70
Are software people treated as professionals within your firm?	85	15
Is your firm actively pursuing some form of software improvement strategy?	68	32

could control it and report its status. We invented terms such as *inch pebbles* to describe the depths to which our plans had to go so management would understand what we wanted.

Today, many firms have placed increased attention on planning, because they are trying to shorten the time-to-market interval by scheduling tasks in parallel using iterative and spiral techniques instead of sequential development approaches. They do so because there isn't enough time to plan all the options. To cope with such contingencies, we have learned that we can do planning in a just-in-time manner. We have also learned that plans should be living documents, iterating and evolving over time.

Sin 3: Unrealistic schedules and budgets

In the past, senior managers often set software budgets and schedules because they did not believe our estimates. We had little data to justify our contention that we just couldn't do what they expected with the resources allocated. In addition, many software managers had no idea what it would take to deliver a quality product. Consequently, these managers often said, "We will try our best," instead of "No! Based on our experience, it will take more resources to do this job." The tables have turned as modern cost-estimating methods and models have emerged.[8]

Today, software managers can say with conviction whether they can do the job with the budget authorized. But they still need better requirements and detailed plans to estimate more precisely. Models need to be calibrated to experience so they can achieve the desired accuracy. Improved management processes are not enough—that's what we need to learn here. To go the next yard, firms must bring in specialized management tools and calibrate them to their experience.

Sin 4: Inadequate controls

A decade ago, software managers rarely understood where they stood relative to their budgets and how long and how much it would take to complete their projects. As I've mentioned, their planning was poor and their schedules and budgets unrealistic. In addition, the tracking and measurement techniques software managers used were in their infancy. Few progress metrics and indicators were reported. Modern inspection techniques and statistical controls were still to come.

Today, only the most forward-looking firms seem to use such techniques. We can't properly control what we haven't properly planned. The easiest way to proceed according to schedule is to lie to ourselves. Again, adopting improved planning and tracking processes will go a long way toward slaying this dragon. But we have also learned that putting modern metrics and measures in place to supplement enhanced controls will improve our capabilities and create an atmosphere of trust with senior management.[9]

Sin 5: Undercapitalization

In the late 1980s, the Japanese startled the software world by introducing the software factory concept. They designed such factories as strategic resources that capitalized on the physical plant employed for software development.[10] In other words, they designed and built their software facilities with software development in mind. Trying to leapfrog the competition, Japanese software developers wired, equipped, and tooled their buildings to optimize software productivity. In making much-needed capital improvements (buying workstations, improving the environment, and so forth), they released money that Corporate America held tightly. In addition, they increased budgets for software tools, especially in firms that had adopted new methods and were embracing process improvement technology.

The software development community has learned that we need to put effort today into developing the infrastructure needed to generate products tomorrow.

Today, most firms are heavily undercapitalized regarding software. The reason is simple. These firms just haven't put the effort into assessing and justifying such investments to their management. Instead, they focus on getting their products out using whatever resources are available. The software development community has learned that we need to put effort today into developing the infrastructure needed to generate products tomorrow.

Sin 6: "We're different" syndrome

In the 1970s and 1980s, software developers tried to get management off their backs by hiding behind their product's intangibility. Too often, when senior managers asked questions, the developers would raise the "snow" level to shorten their "do list" and alleviate pressure to increase progress reporting. Clearly, exploiting our mystique this way was a mistake. Senior managers felt better when we told them that we could use what many considered to be best-management practices to report the status of our projects and track their progress. Senior managers also started promoting software people to positions of power when they felt that software managers could deliver what they promised on schedule.

Today, it seems that we have reverted to hiding behind the "we're different" syndrome, again to get management off our backs, especially when schedules are aggressive. Experience shows that it's better to focus on explaining why software is no different than other technical disciplines.

Sin 7: Lack of focus on quality

During the 1970s and 1980s, we had problems with software quality. Many software product releases were buggy. Even worse, consumers often didn't trust software to perform as advertised. Today's software products appear to be no better. Just look at industry-leading browsers and operating systems. Heavy use of commercial off-the-shelf packages, generators, and user application programs complicate the mix even further. In the early 1990s, when Total Quality Management was king, we learned *that nobody* prospers when defective products are released. As more systems come to depend on software, perhaps we should pay more attention to this lesson. Our customers and users demand that we do a better job when it comes to quality.

I hope that you can use these lessons learned to avoid making one or more of these seven deadly software management sins. When you think about it, most of these lessons convey nothing more than common sense. When I echoed these sentiments to several members of the discussion group, the few who responded agreed with me somewhat. I would be interested in receiving your opinions as well.

References

1. M.C. Paulk, D. Goldenson, and D.M. White, *The 1999 Survey of High Maturity Organizations*, Special Report CMU/SEI-2000-SR-002, Software Eng. Inst., Carnegie Mellon Univ., Pittsburgh, 2000.
2. W.S. Humphrey, *Managing the Software Process*, Addison-Wesley, Reading, Mass., 1989.
3. L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, Addison-Wesley, Reading, Mass., 1998.
4. D.J. Reifer, *Practical Software Reuse*, John Wiley & Sons, New York, 1997.
5. D.J. Reifer, "The Seven Sins of Software Management," *Software Summit Proc.*, Defense Information Systems Agency, Washington, D.C., 1993.
6. B.W. Boehm, *Guidelines for Life Cycle Objectives (LCO) and the Life Cycle Architecture (LCA) Deliverables for Model-Based (System) Architecting and Software Engineering*, Univ. Southern California, Los Angeles, 2000.
7. P. Kruchten, *The Rational Unified Process*, Addison-Wesley, Reading, Mass., 1998.
8. B.W. Boehm et al., *Software Cost Estimation with COCOMO II*, Prentice-Hall, Upper Saddle River, N.J., 2000.
9. Software Productivity Consortium, *The Software Measurement Guidebook*, Int'l Thomson Computer Press, Stamford, Conn., 1995.
10. M.A. Cusumano, *Japan's Software Factories*, Oxford Univ. Press, New York, 1991.

Donald J. Reifer is a teacher, change agent, consultant, contributor to the fields of software engineering and management, and author of *Tutorial on Software Management, Fifth Edition*. He is president of Reifer Consultants and serves as a visiting associate at the Center for Software Engineering at the University of Southern California. He is also a member of the *IEEE Software* Editorial Board and editor of its Manager column. Contact him at d.reifer@ieee.org.

Principles of Software Engineering Project Management

Donald J. Reifer

This paper communicates 14 principles of software engineering project management that are based upon the experience of seasoned software managers. To make these principles useful, each is related to the primary functions software managers perform. These principles are based on the fundamental premise that good engineering and classical project management methods, tools, and techniques can be applied in a cost-effective manner to cope with the challenges associated with delivering high-quality software products on schedule and within budget.

Introduction

This article introduces you to 14 principles of software management that revolve around the five primary functions software project managers perform to get their job done effectively: planning, organizing, staffing, directing, and controlling. Software managers must develop skills, knowledge, and abilities in each of these functions to successfully deliver an acceptable product on schedule and within budget. They start by planning their projects thoroughly and creating the road map that they will use to create baselines and expectations. Then they create organizations, staff them with the right mix of talent and capability, develop teams and teamwork, and motivate and direct their human resources toward achieving project-related goals. They have to integrate the work of many participants to pull the pieces of this puzzle together in such a manner that they can achieve aggressive budgets and schedules. They put into place controls and use them to track status and determine whether or not their teams are making suitable progress. They manage risk and deal with the day-to-day issues that can, if left unchecked, impede progress. They replan, refocus, and reenergize their teams as they take detours to overcome the obstacles that get in the way of achieving their goals. Software project managers are focused on delivering acceptable products on schedule and within budget. They intentionally avoid doing things that would distract them from accomplishing this goal.

Tutorial Organization

This article discusses the organization of this seventh edition of the tutorial and provides you with an overview of its contents. The volume is organized in 14 chapters and three appendices. As illustrated in Figure 1, the first three chapters in this volume provide you with necessary background for this tutorial and acquaint you with the topics of software life cycle models and process improvement. The next chapter introduces you to the discipline of software project management and its tools of the trade. The five chapters that follow discuss the five primary functions of software engineering project management: planning, organizing, staffing, direction, and visibility and control. Three additional chapters have been added over the years as extensions to the original material for the following topics: software estimating, risk management, and metrics and measurement. The final two tutorial chapters discuss areas of special interest to software managers: acquisition management and emerging management topics. Three appendices round out the volume. The glossary defines management terms used throughout the tutorial, and the annotated bibliography provides you with recommended readings to amplify the key points addressed within this volume. The final appendix provides biographies of the twelve authors who contributed original works to this edition of the tutorial.

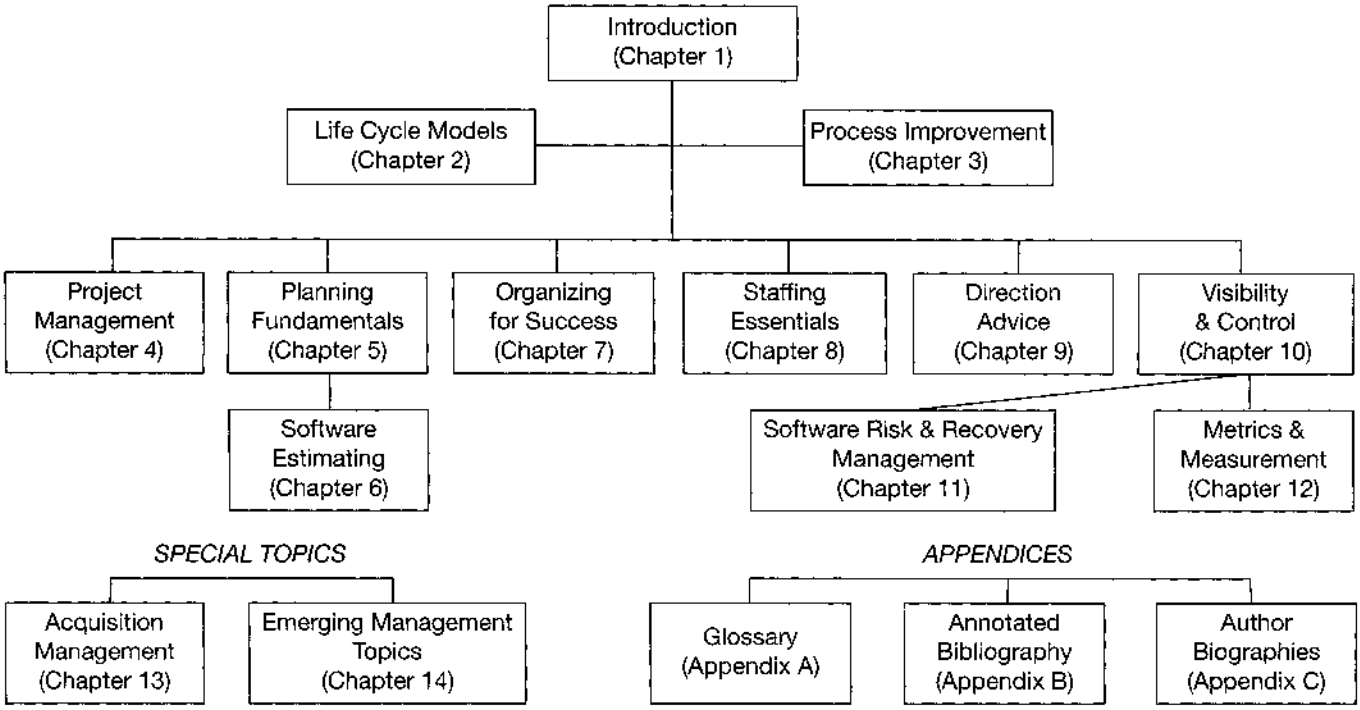


Figure 1. Software Management Tutorial Organization

Planning

Planning is defined as deciding in advance what has to be done, when and how to do it, and who should do it. It encompasses many related disciplines, such as estimating, budgeting, and scheduling. Software managers get involved in many types of planning exercises. For example, they plan projects, capital acquisitions, and/or training/skill development. As discussed in Chapter 5 of this tutorial, plans form the basis against which schedule and budgetary performance are assessed and project control is implemented. Plans create the foundations that project managers use to gain visibility into and control over progress. Based upon the experience of seasoned managers and the software project management body of knowledge,[1] the following three principles establish the basis for project planning:

- **Principle 1—Planning Takes Precedence.** Planning logically takes precedence over all other management functions. While often difficult and time-consuming to perform, plans form the basis for all future work. Managers are encouraged to devote the time needed to figure out what needs to be done, when to do it, who should do it, and how to address the contingencies. Budgets are financial plans, whereas schedules create a viable project time line.
- **Principle 2—Effective Plans Tap the Infrastructure.** Plans are most effective when they are consistent with policies and take full advantage of the organization's existing management infrastructure. This is especially true for those organizations that have initiated a software process improvement program aimed at institutionalizing a "preferred process" and "best practices" across groups.
- **Principle 3—Plans Should Be Living Documents.** Plans should be maintained as living documents or they will quickly become outdated and lose their value as control tools. Plans need to be periodically updated to add detail and reflect the current situation.

Because of their short timetable, most project plans tend to be tactical (near-term) instead of strategic (long-term). When capital and research budgets are impacted, this is a mistake because implementation of project plans may become dependent on others for their realization. The most basic elements of a project plan are its budgets (financial plans) and schedules (delivery timetables). The parts of these documents with the highest leverage are the risk management and contingency plans. It is not uncommon for a manager to spend as much as fifty percent of his/her time early in the project on planning. The better the

plans, the more visibility and control you have over task progress. In addition, the higher in management you move, the more strategic your plans become. For example, product line or line-of-business managers generate product plans whose horizons may be 10 to 20 years long. Independent of the level of planning, each plan you develop represents a guide to some future course of action.

To establish a budget, you will have to prepare a resource estimate (time, staff, budget, etc.) based upon your understanding of what the work is that must be done and the resources you have available to handle the job. The ability to estimate resources accurately is a skill every software manager must possess. As discussed in Chapter 6, poor resource estimates can lead to problems that no amount of dedication, perseverance and hard work can correct. The more accurate the estimate, the higher your chances of success will be.

Organizing

Managers create organizations to achieve their goals and get the work they are responsible for done as efficiently as possible. Such organizations provide a structure that lets managers get work done by assigning responsibilities, delegating authority, and holding people accountable for results. Most managers work within an existing organizational structure. As discussed in Chapter 7, their function is to build teams, staff them, provide them with leadership, direct them, integrate their results, and manage communications up, down, and across the organization. Based upon extensive software engineering project management experience, these tasks lead to the following two organizational principles:

- **Principle 4—Assign Your Software Manager Early.** Recruit your software manager early in the project and empower this professional to perform all of the tasks for which he/she will be held responsible. Ensure that this person occupies a high enough position in the hierarchy to successfully compete for needed resources (staff, budget, etc.), talent, and management support.
- **Principle 5—Give Authority Commensurate with Responsibility.** Your software manager's responsibility should be commensurate with his/her authority. Because software managers are not always masters of their own destiny, they should not be held accountable for results when others' actions impede their performance. Such problems often occur when dealing with requirements. Software managers are not in charge of requirements, but rely on them to form the foundation of their architecture and design.

Many of the organizational factors that impact the performance of software managers fall outside their sphere of control. For example, marketing is often responsible for developing requirements (and their frequent change) and for customer liaison. Influence is the key for gaining control over this untenable situation. The software manager must be able to communicate effectively with others within, across, and up and down the existing organizational framework. As discussed in Chapter 7, working groups and use of integrated product teams are mechanisms that can be used specifically for this purpose.

Communications must flow up and down and across the organization to keep people informed. Staff members must keep abreast of current events or they will lose focus and their performance will be negatively impacted. Newsletters, colloquiums, brown-bag lunches, weekly team meetings and monthly "all hands" meetings, are proven mechanisms for improving communications. They should be exploited, along with software peer reviews and inspections.

Staffing

Staffing refers to recruiting, appraising, growing, and keeping the people you need to get the job done properly. Organizations are only as good as the people who populate them. As discussed in Chapter 8, software managers must be able to recognize talent, breed competence, and weed out deadwood. They must also be able to attract the right people to fill key slots within their organizations. Based upon a great deal of software engineering project management experience, the following two principles establish the basis for staffing:

Principle 6—Care about Your People. Software managers must be able to show their people that they truly care about them, their careers, and their goals. Because actions speak louder than words, managers must demonstrate their devotion by fighting for promotions, salary increases, and better working conditions for their people. They must also be able to coach poor achievers and be able to improve their job-related performance.

Principle 7—Provide Dual-Career Ladders. Promotion should be possible up either a technical or managerial career lad-

der. Technical people who do not want to move into management slots should be given the opportunity to follow other career paths. Chief software engineer positions that are equivalent to middle- and upper-level management slots should be made a visible part of the organizational chart.

In many organizations, dual-career paths act as powerful incentives for technical people who may or may not want to move into management. Knowing what is required to progress along dual lines provides software people with the growth and opportunity they desire. It acts as a motivator and stimulates high levels of achievement. It also makes career counseling easier, especially when people are not satisfied.

As in many other scientific disciplines, good technical performers are often promoted prematurely to management positions. This is frequently a mistake because the skills required for management differ from those required for engineering. Good software managers have to be developed. Training must be provided for those who have demonstrated management potential. Mentoring and other forms of coaching need to be available to help develop individual skills and abilities. In addition, new supervisors should be taught the fundamentals of management.

Directing

Managers get things done through the actions of others. They communicate their goals and lead and motivate their subordinates to achieve these goals typically under deadline pressures. Direction tends to be difficult because software people are highly creative and individualistic. As discussed in Chapter 9, leadership and direction are needed to eliminate mistrust and provide focus for work activities. Based upon lessons learned managing software projects, the following three direction principles create a foundation for our discussions on direction:

- **Principle 8—Provide Your People with an Opportunity to Excel.** Interesting work and the opportunity to excel will motivate your people to do the best that they can. Software managers need to understand how to channel behavior so that it is directed towards achieving work-related goals.
- **Principle 9—Lead by Satisfying Goals.** People will follow those individuals who lead by example and represent a means to satisfy their own personal goals. Success will come to those managers who can make satisfying personal and work-related goals possible on the job.
- **Principle 10—Keep Key People Focused.** Avoid giving your best people too much to do. Too many diversions causes a loss of efficiency that talent alone cannot correct. Learn to say "no" to distractions. To succeed, keep your people focused on the task at hand.

We would like to populate our organizations with talented, self-motivated professionals. Under such a system, tasks would be completed with little or no management interference. Unfortunately, such situations do not exist in most firms. Managers, like coaches, must build synergistic teams and motivate their players to perform at their fullest capability. Managers must be able to communicate, lead, and motivate the players so that they can survive the trials of combat. In addition, they must be able to focus the team, as necessary, to meet deadlines.

Controlling

Planning and control tend to be closely related activities. Managers control by tracking progress against plans and acting on observed deviations. They track actuals against targets and forecast trends. Controls should be diagnostic, therapeutic, accurate, timely, understandable, and, most important, economical. They should call attention to significant deviations from the norm and suggest ways of fixing the problems. They should be forward-looking and emphasize what you need to do in the future to make corrections relative to your plans.

Controls that should be imposed throughout the software development process are discussed in Chapter 10. To be in control, managers must manage risks (see Chapter 11) and use metrics to manage (see Chapter 12). Based upon extensive project management experience, this leads to the following three control principles:

- **Principle 11—Focus on the Significant Deviations.** Controls should be implemented to alert managers promptly to significant deviations from plans. The philosophy of "if it isn't broke, don't fix it" should be remembered. In other words, don't interfere if things are going well and the prognosis looks good for the future.

- **Principle 12—You Cannot Control What You Cannot Measure.** Effective control requires that we measure performance against standards. Normally, these standards are budgets and schedules established as targets within your project plan. However, other standards may exist, especially when metrics and other indicators are used to track status and measure progress. Independent of the system you use, you cannot determine where you are going if you do not know where you are or have been.
- **Principle 13—Make Risk Abatement Your Goal.** Risk management and abatement must be an integral part of any control system or else it will cease to work. Identifying obstacles and figuring out ways to avoid them in advance is an essential part of the control process.

Controls close the loop in the feedback system. They provide managers with the visibility and insight needed to make better and timely decisions. As noted in Chapters 10 to 12 of this tutorial, software managers rely on configuration management, metrics and measurement, quality assurance, software inspections, risk management, and verification and validation techniques to provide them with visibility into the project's status and control over its progress.

Instituting Technology Change

The software industry is in a constant state of change. Managers need to be aware of advances that are being made in order to harness them for their benefit. Although using new technology may lie risky, software managers must be able to figure out when and how to put it to work for their organization's benefit. Otherwise, the ability to get the job done may he hindered. Some of the emerging management concepts are discussed in Chapter 14. Experience with such new technology gives rise to the following final principle of software project management:

- **Principle 14—Match Technology Risk with Expected Benefits.** Technology should be used only when the risk associated with its use is acceptable. For projects on a tight schedule, the introduction of something new may be unacceptable. Yet, the same technology may be defensible on another project where adequate resources are available to insert it operationally.

I Firmly believe that technology transfer is the primary means we have to alleviate most of the software problems the industry is experiencing. We need to figure out how to tap the benefits of new technologies, like those explained in Chapter 14, without paying too high a price. We need to work smarter and harder, or we may not be able to handle the workload in the future.

Summary and Conclusions

I am indebted to many good managers with whom 1 have had the pleasure of working over the past thirty years. They have taught me a great deal. Their conduct has influenced my conduct. Their wisdom has made me wiser. Their experience has become a part of mine. They have provided me with role models, mentored and coached me, spurred me on when I needed motivation, and influenced my management style. My goal with this tutorial is to help you improve your management capabilities by communicating the lessons 1 have learned primarily from others in the form of the 14 principles of software management that I used to organize and shape this volume.

1 would like to thank the IEEE Computer Society for motivating me to keep this tutorial current. The field of software management has made great strides since the last edition of this tutorial. I hope the next few years will see us make even more progress.

Final Thoughts

It is interesting to reflect back 27 years when this tutorial was first published. At that time, I was teaching project management and could not find a suitable text. 1 wrote the first edition to fill that gap. Those universities and colleges that adopted this book as their text told me that they also could not find a suitable textbook. Now. when I ask my professor Friends why they are

still using my text, they give a different reason. They say that there just are too many texts available. They use this tutorial because it distills the body of knowledge in software project management down to the point where it can be taught. It is nice to see such progress.

Reference

1. Project Management Institute, *A Guide to the Project Management Body of Knowledge*, 2000.

The "3 P's" Of Software Management

Donald J. Reifer

Abstract. This paper discusses the "3 P's" of software management, the principles devised to discipline the *processes,* standardize the *products,* and professionalize the *people* you use to generate software within your organization. This paper discusses each of the 3 P's and shows how each is related to the others and the methods, tools, and techniques you use to direct and control your software projects.

Introduction

Producing a large software system is fraught with the problems inherent in any highly labor-intensive activity. A large workforce must be assembled and organized into teams. The engineering and management processes must be defined and put into place before the work starts. A software engineering environment and associated tools must be acquired to support selected methods and to automate tedious tasks. Requirements must be specified and the customer's expectations must be known. Plans must be developed, tasks and related milestones must be defined, and budgets and schedules must be negotiated and agreed upon by the key players. A variety of controls must be put into place so that you can assess your progress, and metrics, reviews, reports, and risk management procedures must be provided. Staff must be brought on, trained and molded into teams to deliver quality products per the negotiated budgets and timetables. People must learn to collaborate, cooperate, communicate, and work together to achieve desired results.

As discussed in the previous paragraph, software project managers must address a wide range of issues when tasked with delivering a "high-quality" solution to the user's needs per an agreed-upon budget and schedule. Their job can be likened to putting a puzzle together while on the deck of a ship that is navigating through a storm. To deliver what is promised, software managers must put into place a management infrastructure that allows them to integrate the products of the many processes that their people use to get their work done. In conjunction with these activities, successful managers must address the variety of financial, social, psychological, political, and technical issues that arise and often impede their progress. To succeed, software managers must perform the following activities:

- Planning
- Organizing
- Staffing
- Directing
- Controlling
- Integrating

Based upon the nature of these activities, I do not believe that novices using recipes and cookbooks can practice software management. Success as a manager requires skill, knowledge, and the ability to get a tough job done under extreme deadline and budgetary pressures. Because managers must direct and coordinate the efforts of teams of highly creative people, their job can he simultaneously challenging and frustrating. By its very nature, the practice of software management requires a great deal of common sense, awareness, and sensitivity. Innovation is necessary at times to cope with the many challenges that arise. Persistence is needed to identify the true causes of their problems. Experience is needed to figure out what has to be done, by whom, and when. Openness to a continual stream of new ideas is a prerequisite for success.

In contrast to the articles written about "software failures," I would like to focus on "software successes." I believe many of us in the industry know how to manage a software project successfully. Unfortunately, because of the pressures to deliver, we do not always put this knowledge into practice when necessary. To help you correct this state of affairs, I have prepared both this book and this paper. The book's primary goal is to pull together the ingredients for success in software project manage-

ment into one place so readers can use it. It provides the body of knowledge that every software manager should know in order to get their job done. I have prepared this paper to introduce you to the concept of the "3 P's of software management." This concept is important because it influenced the selection of papers for this volume. I have also prepared a follow-on paper entitled "Principles of Software Engineering Project Management" (see page 9) to introduce you to the structure and contents of the book.

Setting the Stage

People have often asked me, "Why have you succeeded as a software project manager when others have failed?" That's not an easy question to answer. After considerable thought, the idea of the "3 P's of software management" struck me. While simple, the idea is profound. I immediately tested the idea against the projects that I had managed, both successful and not. I became convinced that the concept had merit because it let me explain to others the attributes of success. What are the 3 P's? Simply stated, they are the *processes, products,* and *people* that populate today's software-intensive projects. To be successful, managers must control the 3 P's concurrently and constantly reconcile conflicts that occur among them.

Think about this concept for a minute. The idea suggests that just concentrating on any one of the 3 P's alone will not be enough to guarantee project success. For example, those who have focused on a process have learned that they need to motivate their people to use it. Just having a good process is not enough. Engineers must be convinced that the process makes their job easier and improves product quality. Otherwise, they will resist using it. However, a process is only one of the contributors to project success. In many organizations, people problems dominate process issues. For example, the pressures associated with deadlines might force key people to quit midway through testing. Social and psychological issues might cause a lack of teamwork just when you cannot afford it. In the product dimension, engineering decisions associated with poor performance might be the most important issue facing management. Using this concept of the "3 P's," you can glean principles that can be used to reconcile the conflicts between differing goals and provide criteria for action. Such principles by design are descriptive, not prescriptive.

Maturing the Process

Producing software involves more than just writing programs. Software should be thought of as a product that must be specified, designed, built, tested, and documented in a disciplined manner. It must be integrated with other products and with the hardware, and you must show your customers that it satisfies their requirements and works as expected in their operational environment. Software product development and maintenance progresses through a series of interrelated, time-phased activities called a life cycle model. Many different life cycle models exist that allow you to sequence activities and order your deliverables. In some, work is done in parallel whenever possible to speed developments and make impossible schedules possible. In others, work is done in spirals to address risk through rapid prototyping. Independent of the model you select, the process you employ must be fully defined, taught, and supported prior to being institutionalized for everyone in your organization to use.

The following three principles are provided to help you decide if the processes you have adopted for your organization are mature and robust enough to satisfy your needs:

- **Principle 1—Recognize that Good Processes Add Value.** Arm good people with processes that they understand and believe in and they will excel. As we have already stated, having either a good process or good people is not enough. You must have both to succeed when faced with the difficulties of generating complex products under budget and schedule pressures. Getting your people to use the process is the challenge. This can be handled most effectively by making your process the preferred way of doing business.
- **Principle 2—Use Your Processes to Share Your Lessons Learned.** Direct your process-definition efforts toward institutionalizing a preferred approach to doing business. This framework represents the scaffolding upon which you will build your management infrastructure (policies, practices, team mechanisms, etc.) and get work done. Such an infrastructure enables you to share your experiences and build on your lessons learned, both positive and negative.
- **Principle 3—Stress Continuous Process Improvement.** Aim your advanced efforts in process development toward using metrics, measurement data and quantitative methods that are directed toward continuously improving your processes. Make sure that the process you improve is the one that your people use. Be flexible and try to build on the past in a manner that lets you address the future. Try to take both people and products into account as you make your decisions based upon the data you collect as part of the normal way that you do business.

To emphasize the progress we have made in the area of process improvement, I have rewritten the chapter on this topic in the current edition of the tutorial. This chapter highlights the experience in using the Capability Maturity Model (CMM) developed at the Software Engineering Institute (SEI) as a process improvement framework. The CMM builds on the five-level process maturity model popularized by the SEI in the late 1980s to characterize the maturity of the practices your people use to generate their software products. The CMM creates a structure that makes insertion of a preferred process into organizations easier to accomplish. This model can be used to rate and rank the "effectiveness" of your current software process and identify shortcomings. Using information from this evaluation, you can implement an improvement program aimed at reaching higher levels of process maturity.

I would suggest that you become familiar with the newer version of the CMM called the CMM Integration (CMMI).[2] This new and more powerful version of the CMM permits you to improve your collective set of engineering and manufacturing practices. The CMMI is beginning to have a profound impact throughout industry because it helps to provide better ways of getting the work done. In addition, models like the CMMI let you both benchmark your process and compare it against your competition. This helps you to identify what practices you should use to develop quality products more quickly and efficiently.

As my principles suggest, emphasis on improving your process is a necessary, but not entirely sufficient, condition for project success. The reason for this is simple. In order to make the process perform as expected within your operational environment, you need to consider both its product and people implications. For example, selection of a process that does not support the specific engineering practices needed for a product could lead to catastrophe. And the process is doomed to failure if the people who are tasked to use it do not believe in its capabilities. Instead of using the process, your people will devote their time and energy to either finding ways of going around the process or justifying why it cannot be used.

Focusing on Product Issues

Let us now look at the issues associated with product management. To be successful, you really need to understand what you are building, why you are building it, and the related building codes. Good processes help you structure how to build; they do not tell you what to build. They do not establish requirements for form, Fit, and function. To build a software product that satisfies your customer, you must put an engineering methodology into place and follow it and its associated building codes to bring high-quality products to market. You must understand both the customer and technical requirements and design a solution that satisfies them. You must establish a product vision and pursue the technology that enables you to realize it now and over time. You must focus on both solidifying your architecture and taking advantage of the opportunities for sharing of components across families of like systems. You must also be able to demonstrate that the product works as specified in an operational setting so that end users will applaud your accomplishments.

The following additional four principles are provided to help you put the different product and product line characteristics into action when you are building software for your systems:

- **Principle 4—Recognize that Performance Is Always the Issue.** Focus your efforts on performance because that is what makes or breaks your product from a customer's point of view. Normally, about twenty percent of the software consumes eighty percent of the computational resources (e.g., memory, speed, I/O capacity). Identify these parts and build them carefully. If you are replacing an existing system, establish a baseline for the current performance. This enables you to make quantitative comparisons and demonstrate in the future that you have achieved your performance goals.
- **Principle 5—Realize that Quality Makes the Difference.** When faced with a choice, your customers will always select quality when functionality and price are nearly equivalent. However, remember that the price/performance trade-off is only important when the product has the features that your customers want and when it works as expected.
- **Principle 6—Emphasize that the Customer is Always Right.** Be customer directed in your functional choices because the customers are ultimately the ones who buy and use your product. Involve them in your process at strategic milestones and do everything you can to tap their knowledge and experience as you build the product. Aim your quality assurance activities at customer satisfaction, not specification conformance.
- **Principle 7—Avoid Gold Plating and Feature Creep.** Avoid gold plating and feature creep at all costs; they will doom your software development project to failure. No matter how hard you try, you cannot deliver an acceptable product on time and for a negotiated price when your requirements are changing and needless additions are being made to the system.

As these principles suggest, marketing rather than engineering issues drive product development. I learned early in my career to pay as much attention to product packaging as I did to content. A feature-rich product will not sell if it is not easy to

understand and use. And the principles of total quality management definitely hold true. Your goal should be to build the right product the first time. To achieve this aim, you must package the features, functionality, and performance that your customers want, need, and expect in an acceptable form. You must also embed quality into the product and make sure it is reliable, maintainable, and usable when it is released to the field. Finally, you must avoid rework and take advantage of opportunities for commonality and reuse to keep your costs down and remain competitive.

Most of my discussion so far has focused on the deliverable products. Yet. there are many nondeliverable engineering products that are generated during the process that deserve attention. Most software engineering efforts generate documentation of some form. All too often, we produce either too little or too much paperwork. Planning is needed to make sure the documents produced are actually used. This discussion leads us to our fifth and final product principle:

- **Principle 8—Eliminate Unnecessary Paperwork.** Do not produce paper for the sake of producing paper. Such folly results in wasted effort. Understand what documentation you need and devise a plan to develop it. Distinguish between deliverable and nondeliverable documents. Make sure that every document you generate serves a worthwhile purpose.

Realize that generating documentation takes time and effort. It also takes effort to review, distribute, manage, and control versions. I suggest that you generate only those documents that are truly needed to service your user, developer, maintainer, management, and customer communities. Realize that documents come in four varieties: specifications, plans, test documents, and manuals. In addition, the code itself should be self-documenting. After all, that is what the programmers really trust and read.

Within the last few years, the concepts of agile methods for software product lines, architectures, and reuse have become popular as has the move to software patterns and components. My chapter on emerging management topics and some of the case studies in other chapters treat these important techniques. For those interested, I have included several agile method papers throughout the text and updated my paper on the topic of product lines in order to summarize where the state of the technology is today and where it is heading. I have also included a similar survey paper in my chapters on technology transfer and metrics and measurement to round out this update to the tutorial volume.

Addressing the People-Oriented Needs

The final "P" revolves around people. They are your most precious resource because it is through their efforts that you as a manager are successful. Good software managers know how to recruit, grow, motivate, and retain the best people. They know how to use the system to recognize and reward their high performers and provide them with interesting work and the opportunity to excel on the job. When their people are enthusiastic, they are smart enough to get out of the way and watch them shine. When their people are unmotivated, they know how and when to step in and take charge of the situation. When good people are involved, the impossible becomes possible. In my opinion, highly motivated, talented people are the key to software success.

As a manager, you know how to get things done through the work of others. Your primary job is to stimulate your staff to do the best that they can commensurate with their abilities. To motivate staff, you must be sensitive to their needs, understand what drives them, trust them implicitly, and he able to lead them. To get your people to work together, you must build teams and facilitate teamwork. For team building, you must stress the importance of collaboration, cooperation and communication as part of the job. To achieve high levels of performance, you must provide your people with a good work environment and a variety of tangible and intangible rewards. You must realize that when your people shine, you will shine. Your outward sign of success will he the "results" your people achieve. You must be able to plan, direct, and integrate the work of others so things will happen in predictable ways.

Good managers can be likened to good coaches. They know how to recruit the right talent, build teams, and get people to perform. Positive results occur on the playing field when things go well. When problems arise, they know when to step in and what to do to develop a remedy.

Again, I have tried to revise this tutorial to include better and more up-to-date papers on the people-oriented software management disciplines of organizing, staffing, and direction. I have also tried to include some case studies to show how to direct behavior when deadlines get tough and people are overworked. I have also developed the following five people-oriented principles to guide your future personnel actions:

- **Principle 9—Reward Your Top Performers.** Know who your top performers are and do everything you can to keep them happy. Realize that as much as eighty percent of the work is done by twenty percent of your staff. Recognize and reward these high producers and make sure they are not stretched too thin.

- **Principle 10—Commit to Personal Growth.** Make an open and visible commitment to staff development and growth. Know your people and help them achieve their personal goals through work-related training, mentoring, and job assignments. If you help people be all they can be, they will follow you anywhere. More important, they will expend whatever effort is needed to get the job done right the first time.
- **Principle 11—Recognize what Motivates Performance.** People will try their best when they are given interesting work, growth opportunities, feedback, praise, recognition for a job well done, and the ability to excel. To motivate your people, you must recognize, respect, and respond to their needs. You must tap potential, deal with personalities, and get your people to try their hardest, especially when the chips are down and the situation seems desperate.
- **Principle 12—Build Bridges through Open Communication.** Most disputes that occur between staff members are prompted by either intolerance or poor communication. To address these problems, you must stimulate a free exchange of information and ideas. You must permit your interdisciplinary and integrated product teams to go across organizational boundaries when necessary. You must set up vertical and horizontal channels to keep pertinent information flowing up, down, and across the organization. Most important, you must stay engaged with your people even when your work load is intolerable.
- **Principle 13—The Equality Principle.** Reward competence and incompetence with equal vigor. Otherwise, you will foster ineptitude and complacency. Help your people set aggressive but realistic goals and then hold them accountable for results. Be fair but austere in your convictions. Coach your staff and show them how to do a good job. After you have exhausted every other avenue, do not be afraid to terminate an employee for poor performance.

As these principles suggest, software managers need to focus more of their attention on the needs of their people. Many of these needs revolve around your people's desire for structure, leadership, coaching, direction, and attention. When your staff knows what is expected, they will achieve their objectives. They want to do a good job. Your job is to make such performance possible. Managers must also arm their people with the processes, tools, and work environment necessary to complete their assignments. They must be able to focus the energy of their teams so that daily distractions do not interfere with completing the task at hand. They must establish a communications infrastructure and do everything possible to get people to use it. Communications problems lead to many of the people problems that I have been exposed to over the years in large software projects.

Software managers often have difficulty in dealing with people issues because they come from technical specialties like engineering, mathematics or the sciences. They must learn to handle the conflicts that arise between logic and emotion fairly and openly. They must also learn to tap the knowledge base of experience that others have developed to manage the work of others. This edition of the tutorial contains several new papers that address the human side of the management process.

Summary and Conclusions

I believe the concept of the "3 P's of software management" is very meaningful. It allows you to separate concerns and address the relationships between the things that make a difference in a software development effort. In essence, the "3 P's" (process, product, and people) act as the three independent variables that you must control in any software development effort. By addressing each of these variables consciously, I believe you can greatly bolster your overall ability to manage either a software development effort or an organization. This paper provides you with principles that allow you to take advantage of the lessons we have learned relative to the "3 P's." These principles are aimed at helping you succeed when you are tasked with delivering an acceptable software product under deadline pressures.

References

1. M. C. Paulk, C. V. Weber, B. Curtis, and M. B. Chrissis, *The Capability Maturity Model: Guidelines/or Improving the Software Process*, Addison-Wesley, 1995.
2. M. B. Chrisis, M. Konrad, and S. Shrum, *CMMI*, Addison-Wesley, 2003.

Why Big Software Projects Fail: The 12 Key Questions

Watts S. Humphrey
The Software Engineering Institute

In spite of the improvements in software project management over the last several years, software projects still fail distressingly often, and the largest projects fail most often. This article explores the reasons for these failures and reviews the questions to consider in improving your organization's performance with large-scale software projects. Not surprisingly, considering these same questions will help you improve almost any large or small project with substantial software content. The principal questions concern why large software projects are hard to manage, the kinds of management systems needed, and the actions required to implement such systems. In closing, the author cites the experiences of projects that have used the methods described and cites sources for further information on introducing the required practices.

Software project failures are common, and the biggest projects fail most often. There are always many excuses for these failures, but there are a few common symptoms. Some years ago, before the invention of the Capability Maturity Model® (CMM®) and CMM Integration℠ (CMMI®) the principal problem was the lack of plans [1, 2]. In the early years, I never saw a failed project that had a plan, and very few unplanned projects were successful.

The methods defined for CMM and CMMI Levels 2 and 3 helped to address this problem. As the Standish data in Figure 1 shows, the success rate for software organizations improved between 1994 and 2000, and much of this improvement was due to more widespread use of sound project management practices [3]. Still, with less than 30 percent of our projects successful, those of us who are software professionals have little to be proud of.

The definition of a successful project is one that completed within 10 percent or so of its committed cost and schedule and delivered all of its intended functions. Challenged projects are ones that were seriously late or over costs or had reduced functions. Failed projects never delivered anything. Figure 2 (see page 26) shows another cut of the Standish data by project size. When looked at this way, half of the smallest projects succeeded, while none of the largest projects did. Since large projects still do not succeed even with all of the project management improvements of the last several years, one begins to wonder if large-scale software projects are inherently unmanageable.

Question 1: Are All Large Software Projects Unmanageable?

There are some large, unprecedented projects that are so risky that they would likely be challenged under almost any management system. But some large projects have succeeded. Two examples are the Command Center Processing and Display System Replacement (CCPDS-R) project, described by Walker Royce, and the operating system (OS)/360 project in my former group at IBM [4, 5]. The CCPDS-R was a U.S. Air Force installation at Cheyenne Mountain in Colorado. It had about 100 developers at its peak. The OS/360 was the operating system to support the IBM 360 line of computers, and included the control program, data management, languages, and support utilities. Its development team consisted of about 3,000 software professionals.

Both of these projects placed heavy emphasis on planning, and both adopted an evolutionary development strategy with multiple releases and phased specifications. Both projects also took a somewhat unconventional approach to motivating team member performance. For CCPDS-R, management distributed 50 percent of the project award fee to the development team members. This built their loyalty and commitment to success, and maintained team motivation throughout the job. The CCPDS-R project was delivered on schedule and within contracted costs.

By the time I took over the OS/360 project some years ago, we had all learned that the proper strategy for building big software-intensive systems was to break the job into as many small incremental releases as practical. Since this strategy required organization-wide coordination, our very first action was to have all the development teams in all the involved laboratories produce their own plans and coordinate them through a central build-and-release group. Then, we based the company's commitments on the dates that the teams provided. In no case did IBM commit to any date that was not supported by a plan that had been developed by the team that was to do the work.

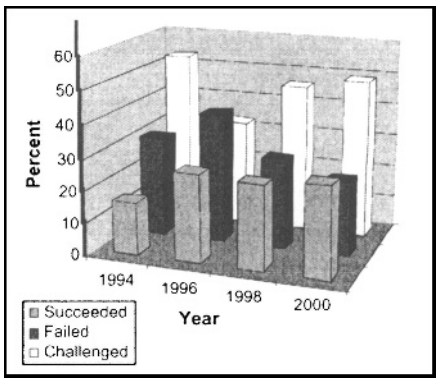
These plans extended through 19 releases over a period of 30 months. Most importantly, they provided the focus we all needed to coordinate the work of 15 laboratories in six countries and to promptly recognize and address the myriad problems that inevitably arose. The developers were personally committed to their schedules, and they delivered every one of these releases on or ahead of the committed schedules. So, at least based on this limited sample, some large software projects can be managed successfully. However, because the success rate is so low, large-scale software projects remain a major project management challenge.

Question 2: Why Are Large Software Projects Hard to Manage?

While large software projects are undoubtedly hard to manage, the key question is "Why?"

Historically, the first large-scale management systems were developed to manage armies. They were highly autocratic, with the leader giving orders and the

Figure 1: *Project Success History [3]*



Reprinted from *CrossTalk*, March 2005.

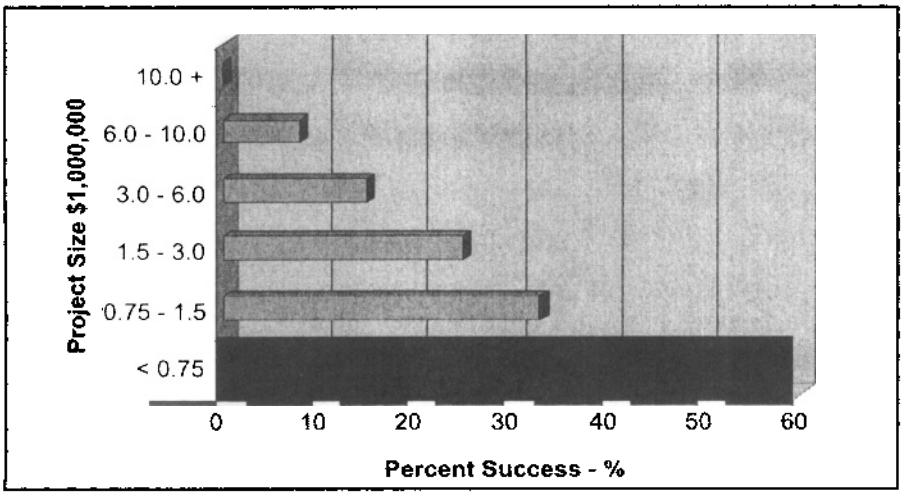
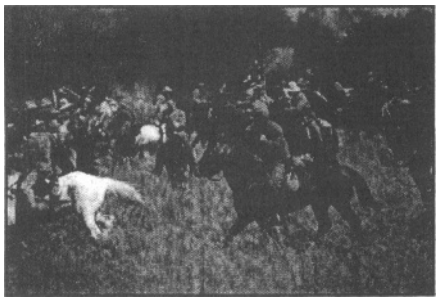


Figure 2: *Success Rate by Project Size [3]*

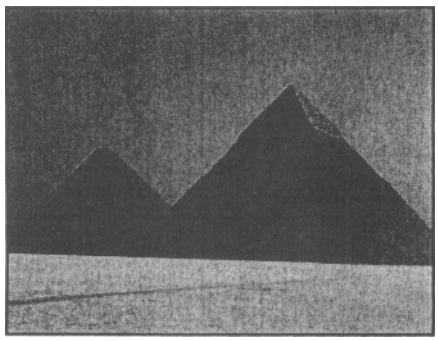
troops following. Over time, work groups were formed for major construction projects such as temples, palaces, fortifications, and roads. The laborers were mostly slaves, and again, the management system was highly autocratic. The workers did what they were told or they were punished.

This army-like structure was essentially the only management system for many years until the Greek city-states introduced democratic political systems. However, these democratic principles were primarily used for governing, not for project management. Somewhat later, a totally different management system was used to build cathedrals. This work was largely done by volunteer artisans who managed themselves under the guidance of a master builder. Since building a cathedral often took 50 years or more, the cathedral-management system is not a good model for modern large-scale software projects. However, it did produce some beautiful results. This cathedral-building management system was not used for anything but cathedrals for many years, but it has recently had some success as the guiding principle for the open source software development community [6].

The first large-scale management systems were developed for armies. Leaders gave orders and troops followed. With training and discipline, this approach could work even amid chaos and confusion.



Large-scale management systems were eventually applied to major construction projects. The system was highly autocratic; workers did what they were told or they were punished.



The next major management innovation was the factory. Factories started producing clothing and were soon used for making all kinds of goods. Again, however, the factory management system was autocratic, with management directing and workers doing. While the factory model improved productivity, it was not without its problems. The early work of Frederick Winslow Taylor about 100 years ago and the more recent work of W.E. Deming, J.M. Juran, and others has improved the effectiveness of this model by redefining the role of the worker. The modern view is that to do quality work for predictable costs and schedules, workers must be treated as thinking and feeling participants rather than merely as unfeeling drudges. However, to date, these methods have had limited application to software [7, 8, 9].

The factory/army system has persisted and now characterizes the modern corporate structure where senior management decides and everybody else follows. Many managers would contend that they listen to their people while making decisions. However, employees generally view corporate management as autocratic and few feel that they could influence a senior manager's decisions. Some managers even argue that autocratic management is the only efficient style for running large projects and organizations. Democratic debates would take too long and decisions would not be made by the most important or knowledgeable people.

Regardless of the validity of this view, the hierarchical management style does not work well for managing large software projects. Unfortunately, except for the cathedral-building system, there is no other proven way to manage large-scale work. So, if we want to have successful large-scale software projects, we must develop a project management system that is designed for this purpose.

Question 3: Why Is Autocratic Management Ineffective for Software?

Before developing a new management system, we should first understand why the current one does not work. To answer this question, we must explore the nature of software work and how it differs from other, more manageable work. Software and software-like work have characteristics that are particularly difficult to manage. From a management perspective, the principal difference between managing traditional hardware projects and modern software work concerns management visibility.

With manufacturing, armies, and traditional hardware development, the managers can walk through the shop, battlefield, or lab and see what everybody is doing. If someone is doing something wrong or otherwise being unproductive, the manager can tell by watching for a few minutes. However, with a team of software developers, you cannot tell what they are doing by merely watching. You must ask them or carefully examine what they have produced. It takes a pretty alert and knowledgeable manager to tell what software developers are doing. If you tell them to do something else or to adopt a new practice, you have no easy way to tell if they are actually working the way you told them to work.

Some might argue that hardware work is not actually that different from software work and that, at least for some hardware tasks and most system engineering jobs, the work is equally opaque to management. This is certainly true, particularly when the hardware engineers are producing microcode, using hardware design languages, or working with simulation or layout tools. Today, as modern technical specialties increasingly overlap, many hard-

ware projects now share the same characteristics as large software projects. When hardware development and system-engineering work have the characteristics of software work, they should be managed like software. However, since these systems groups generally tend to be relatively small, they do not yet present the same project-manageability problems as large-scale software.

Question 4: Why Is Management Visibility a Problem for Software?

Since most software developers are dedicated and hard-working professionals, why is management visibility a problem?

The problem is that the manager cannot tell where the project stands. To manage modern large-scale technical work, you must know where the project stands, how rapidly the work is being done, and the quality of the products being produced. With earlier hardware-development projects, all of this information was more-or-less visible to the manager, while with modern software and systems projects it often is not.

This is a problem because large development projects, whether hardware or software, always run into problems, and every problem involves more work. While developers can invariably overcome small problems, every problem adds to the workload and delays the job. Each little slip is generally manageable by itself, but over time, problems add up, and sooner or later the project is in serious trouble.

The project manager's job is to identify these small daily slips and to take steps to counter them. As Fred Brooks said, "Projects slip a day at a time" [10]. With traditional hardware projects, the manager could usually see these one- and two-day slips and could do something about them. With modern, complex, software-intensive systems, the daily schedule slips are largely invisible. So, with large-scale software work, the managers generally do not see the schedule problem until it is so big that it is obvious. Then, however, it is usually too late to do much about it.

Question 5: Why Can't Managers Just Ask the Developers?

If the managers cannot see where the developers stand, why not just ask them?

Most developers would be glad to tell their managers where they stood on the job. The problem is that, with current software practices, the developers do not know where they stand any more than the managers do. The developers know what they are doing, but they do not have personal plans, they do not measure their work, and they do not track their progress. Without these practices to guide them, software people do not know with any precision where they are in the job. They could tell the manager that they are pretty close to schedule or 90 percent done with coding, but the fact is that they do not really know. Again, as Brooks said, "...programmers generally think that they are 90 percent through with the coding for more than half of the project" [10].

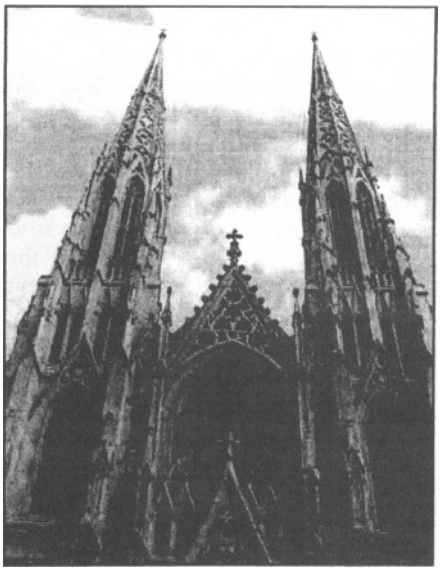
Unless developers plan and track their personal work, that work will be unpredictable. Furthermore, if the cost and schedule of the developers' personal work is unpredictable, the cost and schedule of their teams' work will also be unpredictable. And, of course, when a project team's work is unpredictable, the entire project is unpredictable. In short, as long as individual developers do not plan and track their personal work, their projects will be uncontrollable and unmanageable.

Anyone who has managed software development will likely argue that this is an overstatement. Although you may not know precisely where each developer's work stands, you can usually get a general idea. Since about a third to a half of the small projects are successful when the developers do not plan and track their personal work, such projects can be managed. So why should the lack of sound personal software practices be a problem for large projects?

It is true that software projects are not totally unmanageable. As Figures 1 and 2 show, the worst problem is with the very large software projects. On small projects, some uncertainty about each team member's status is tolerable. However, as projects get bigger and communications lines extend, precise status information becomes more important. Without hard data on project status, people communicate opinions, and their opinions can be biased or even wrong. When filtered through just a few layers of management, imprecise project status reports become so garbled that they provide little or no useful information. Then these large-scale software projects end up being run with essentially no management visibility into their true status, issues, and problems.

Question 6: Why Do Planned Projects Fail?

Today, with CMM and CMMI, most large software projects are planned, and they use methods like Program and Evaluation and Review Technique (PERT) and earned value to track progress. Why is that not adequate?



Work on cathedrals was done by volunteer artisans who managed themselves under the guidance of a master builder. This approach has had some success in open source software development.

The problem is with the imprecision and inaccuracy of most software project plans. Most projects have major milestones such as specifications complete, design complete, code complete, and the like. The problem is that on real software projects, few of these high-level tasks have crisp completion dates. The requirements work generally continues throughout design and even into implementation and test; coding usually starts well before design completion and continues through most of testing.

A few years ago, the management of a large software organization asked me to review their largest project. They told me that the code completion milestone had already been met on schedule. However, I found that very little code had actually been released to test. When I met with the development teams, they did not know how much code they had written or what remained to be done. It took a full week to get a preliminary count, and it was a month before we got accurate data. It was another 10 months before all of the coding was actually completed. It is not that developers lie, just that without objective data, they have no way to know precisely where they stand. When they are under heavy schedule pressure, people try to respond. Since we all know that the bearer of bad news tends to be blamed, no one dares to question the schedule and everyone gives the most optimistic story they can.

Question 7: Why Not Just Insist on Detailed Plans?

Why cannot management just insist on more detailed plans? Then they could have more precise measures of project status.

While this would seem reasonable, the issue is, "Whose plans are they?" Detailed plans define precisely how the work is to be done. When the managers make the plans, we have the modern-day equivalent of laborers building pyramids. The managers tell the workers what to do and how to do it, and the workers presumably do as they are told.

While this has been the traditional approach for managing labor, it has become progressively less effective for managing high-technology work, particularly software. The principal reason is that the managers do not know enough about the work to make detailed plans. That is why many of these software-intensive projects typically have very generalized plans. This provides the developers with the flexibility they need to do creative work in the way that they want to. The current system is therefore the modern equivalent of the cathedral-building system where the developers act like artisans. The unfortunate consequence is that, without Herculean effort, it often seems that the natural schedule for such projects could easily approach 50 years.

Question 8: Why Not Tell the Developers to Plan Their Work?

The obvious next step would be to tell the developers to make their own detailed plans. Why would this not work?

There are three problems. First, most developers do not want to make plans; they would rather write programs. They view planning as a management responsibility. Second, if you told them to make plans, they would not know how to do it. Few of them have the skill and experience to make accurate or complete plans. Finally, making accurate, complete, and detailed plans means that the developers must be empowered to define their own processes, methods, and schedules. Few managers today would be willing to cede these responsibilities to the software developers, at least not until they had evidence that the developers could produce acceptable results.

Question 9: How Can We Get Developers to Make Good Plans?

It seems that the problem of effectively managing large software projects boils down to two questions: How can we get the software developers and their teams to properly make and faithfully follow detailed plans, and how can we convince management to trust the developers to plan, track, and manage their own work?

To get the developers to make and follow sound personal plans, you must do three things: provide them with the skills to make accurate plans, convince them to make these plans, and support and guide them while they do it.

Providing the skills is just a question of training. However, once the developers have learned how to make accurate plans and to measure and track their work against these plans, they usually see the benefits of planning and are motivated to plan and track their own and their team's work. So, it is possible that developers *can* be taught to plan and, once they learn how, they are generally willing to make and follow plans [11].

Question 10: How Can Management Trust Developers to Make Plans?

This is the biggest risk of all: Can you trust developers to produce their own plans and to strive for schedules that will meet your objectives?

This question gets to the root of the problem with autocratic management methods: trust. If you trust and empower your software and other high-technology professionals to manage themselves, they will do extraordinary work. However, it cannot be blind trust. You must ensure that they know how to manage their own work, and you must monitor their work to ensure that they do it properly. The proper monitoring attitude is not to be distrustful, but instead, to show interest in their work. If you do not trust your people, you will not get their whole-hearted effort and you will not capitalize on the enormous creative potential of cohesive and motivated teamwork. It takes a leap of faith to trust your people, but the results are worth the risk.

Question 11: What Are the Risks of Changing?

Every change involves some risk. However, there is also a cost for doing nothing. If you are happy with how your large software projects are performing, there is no need to change. However, few managers or professionals are comfortable with the current state of software practice, particularly for large-scale projects. So, there are risks to changing and risks to not changing. The management challenge is to balance these risks before deciding what to do.

There are two risks to changing to a new management system for large-scale software projects. First, it costs time and money to train the developers to plan and track their work and to train the managers to use a new management system. Then comes the risk of using these methods on a real project. While you will see some early benefits, you will not know for sure whether this new management system is truly effective for you until the first project is completed and you can analyze the results.

This brings up a related and even more difficult problem: On large multi-year projects, there is not time to run pilots. You must pick a management strategy and go with it. However, since almost all large software-intensive projects are now failing anyway, the biggest risk is *not changing*. Perhaps the biggest shock for most managers is realizing that they are part of the problem, and that they have to change their behavior to get the kind of large-system results they want.

These problems are common to all change efforts. The way to manage these problems is to examine the experiences of others and to minimize your exposure by carefully planning your change effort and getting help from people who have already used the methods you plan to introduce. Of course the alternative is to hope that things will get better without any changes. With this choice, however, your large-systems projects will almost certainly continue to perform much as they have in the past.

Question 12: What Has Been the Experience So Far?

The Software Engineering Institute (SEI℠) has developed a method called the Team Software Process℠ (TSP℠) that follows the concepts described in this article [11]. With the TSP℠, if you properly train and support your development people and if you follow the SEI's TSP introduction strategy, your teams will be motivated to do the job properly. The team members' personal practices will be defined, measured, and managed; team performance will also be defined, measured, and managed; and the project's status and progress will be precisely reported every week. Although this will not guarantee a successful project, these practices have worked for the several dozen projects that have tried them so far.

Moreover, there is one caveat. These

practices have proven effective for teams of up to about 100 members, as well as for teams composed of multiple hardware, systems, and software professionals. They have even worked for distributed teams from multiple geographic locations and organizations. Although these methods should scale up to very large projects, the TSP has not yet been tried with projects of over 100 professionals. I know from personal experience, however, that these practices will address many of the problems faced by the managers of software organizations of several thousand developers.

The other articles in this issue describe the TSP experiences of several organizations. They describe how these practices have worked on various kinds of projects and how they could help your organization.◆

Acknowledgements

Many people have participated in the work that led to this article, so I cannot thank them all personally. However, without their willingness to try new methods and to take the risks that always accompany change, this work would not have been possible. So, to everyone who participated in the early CMM and CMMI work and to all of those who have learned and used the Personal Software Process℠ and TSP, you have my profound gratitude. I have also had the advice and support of several people in writing this article. My special thanks go to Dan Burton, Noopur Davis, Bill Peterson, Marsha Pomeroy-Huff, and Walker Royce.

References

1. Humphrey, Watts S. Managing the Software Process. Reading, MA: Addison-Wesley, 1989.
2. Chrissis, Mary Beth, Mike Konrad, and Sandy Shrum. CMMI – Guidelines for Process Integration and Process Improvement. Reading, MA: Addison Wesley, 2003.
3. The Standish Group International, Inc. Extreme Chaos. The Standish Group International, Inc., 2001.
4. Royce, Walker. Software Project Management, A Unified Framework. Reading, MA: Addison-Wesley, 1998.
5. Humphrey, Watts S. "Reflections on a Software Life." In the Beginning, Recollections of Software Pioneers. Robert L. Glass, Ed. Los Alamitos, CA: IEEE Computer Society Press, 1998.
6. Raymond, Eric S. The Cathedral and the Bazaar. Cambridge, MA: O'Reilly Publishers, 1999.
7. Deming, W. Edwards. The New Economics for Industry, Government, Education. 2nd ed. The MIT Press, Cambridge, MA, 2000.
8. Juran, J.M., and Frank M. Gryna. Juran's Quality Control Handbook, Fourth Edition. New York: McGraw-Hill Book Company, 1988.
9. Taylor, Frederick Winslow. The Principles of Scientific Management. New York: Harper and Row, Publishers, Inc., 1911.
10. Frederick P. Brooks. The Mythical Man-Month. Reading, MA: Addison Wesley, 1995.
11. Humphrey, Watts S. Winning With Software: An Executive Strategy. Reading, MA: Addison-Wesley, 2002.

Note

1. The Software Engineering Institute offers courses and transition services to help organizations introduce the TSP. Additional information is available at <tsp@sei.cmu.edu> or at <www.sei.cmu.edu/tsp>.

About the Author

Watts S. Humphrey joined the Software Engineering Institute (SEI℠) of Carnegie Mellon University after his retirement from IBM in 1986. He established the SEI's Process Program and led development of the Software Capability Maturity Model®, the Personal Software Process℠, and the Team Software Process℠. During his 27 years with IBM, he managed all IBM's commercial software development and was vice president of Technical Development. He holds graduate degrees in physics and business administration. He is an SEI Fellow, an Association for Computing Machinery member, an Institute of Electrical and Electronics Engineers Fellow, and a past member of the Malcolm Baldrige National Quality Award Board of Examiners. He has published several books and articles and holds five patents.

Carnegie Mellon University
4500 Fifth AVE
Pittsburgh, PA 15213-2612
Phone: (941) 924-4169
Fax: (941) 925-1573
E-mail: watts@sei.cmu.edu

Software projects are still late, over budget, and unpredictable. Sometimes the entire project fails before ever delivering an application. This clear, commonsense review of fundamental project management techniques reminds us that we still have a long way to go.

Critical Success Factors In Software Projects

John S. Reel, Trident Data Systems

Throughout the fifty-odd years of software development, the industry has gone through at least four generations of programming languages and three major development paradigms. We have held countless seminars on how to develop software correctly, forced many courses into undergraduate degree programs, and introduced standards in our organizations that require specific technologies. Still, we have not improved our ability to successfully, consistently move from idea to product. In fact, recent studies document that, while the failure rate for software development efforts has improved in recent years, the number of projects experiencing severe problems has risen almost 50 percent.[1] There is no magic in managing software development successfully, but a number of issues related to software development make it unique.

Reprinted from *IEEE Software*, May/June 1999.

Managing Complexity

Several characteristics of software-based endeavors complicate management. First, software-based systems are exceptionally complex. In fact, many agree that "the basic problem of computing is the mastery of complexity."[2] Because software developers must deal with complex problems, they are generally very intelligent and complex individuals, which also complicates the management formula. Add the fact that developers are trying to hit a moving target—user requirements—and you get a volatile mixture of management issues.

These and many other influences contribute to a fantastically high failure rate among software development projects. The Chaos study, published by the Standish Group, found that 26 percent of all software projects fail (down from 40 percent in 1997), but 46 percent experience cost and schedule overruns or significantly reduced functionality (up from 33 percent in 1997).[1] The study also shows that the completion rate has improved because companies have trended towards smaller, more manageable projects—not because the management techniques have improved. Can you imagine a construction firm completing only 74 percent of its buildings and completing only 54 percent of the buildings within schedule and budget? To change this trend, we must place special emphasis on certain factors of the management process.

You may think the answers lie in elaborate analysis methodologies, highly advanced configuration management techniques, or the perfect development language. Those elements of the technology landscape are as important as highly scientific and analytical research in analysis and design methodologies, project management, and software quality. However, blueprints of the latest train technology didn't improve life in the Wild West until rail companies invested in the fundamental aspects of train transportation—tracks and depots. Likewise in software, more "advanced" technologies are far less critical to improving practice than embracing what I believe are the five essential factors to managing a successful software project:

1. Start on the right foot.
2. Maintain momentum.
3. Track progress.
4. Make smart decisions.
5. Institutionalize post-mortem analyses.

Granted, even a detailed review of these may leave you wondering what's new here. Not much—this is common-sense, basic management stuff. And yet these principles are not commonly employed. If they were, we would not see such high failure rates.

Start on the Right Foot

It is difficult to call any of these factors most important, since they are all critical to the success of large development efforts. However, getting a project set up and started properly certainly leads this class of factors. Just as it is difficult to grow strong plants in weak soil, it is almost impossible to successfully lead a development effort that is set up improperly. Tom Field analyzed pitfalls in software development efforts and gave 10 signs of IS project failures—at least seven of which are fully determined before a design is developed or a line of code is written.[3] Therefore, 70 percent of the dooming acts occur before a build even starts.

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

Here are 10 signs of IS project failure:[3]

1. Project managers don't understand users' needs.
2. The project's scope is ill-defined.
3. Project changes are managed poorly.
4. The chosen technology changes.
5. Business needs change.
6. Deadlines are unrealistic.
7. Users are resistant.
8. Sponsorship is lost.
9. The project lacks people with appropriate skills.
10. Managers ignore best practices and lessons learned.

Given this information, what can we do to get projects off to a successful start?

Set realistic objectives and expectations—for everyone

The first objective in getting a project off to a good start is to get everyone on the same wavelength. Management, users, developers, and designers must all have realistic expectations. In

case your customers haven't heard, remind them routinely that this system will not solve all of their problems and it will probably create new issues. The new system should cost-effectively solve more problems than it creates. The developers must also understand that the customers do not know exactly what they want, how they want it, or how it will help them. Often, they don't even know how much they can spend. Everyone has to come to the table with their eyes open, willing to cooperate and listen. To avoid later heartache, pay strict attention to the commitments made by both sides.

By the time you figure out you have a quality problem, it is probably too late to fix it.

Build the right team

Next, you must put together the right team. First ensure that you have enough resources to get the job done. If you do not get commitments for resources up front, the effort is doomed. If management is not excited enough about the effort to give it enough resources, you may not have the support necessary for success. Remember, too, that you will likely need more resources than you think. We are all inherently optimistic, so guard your personnel projections and err on the high side from the start.

Building the right team means getting good people. This is hard because companies usually want to place personnel moving off other efforts. Sometimes these people are good resources, but not always. However, also recognize that you do not need, or want, all of the very best designers and developers. In my experience, staffing around 20 percent of the team with the best available works well. This figure is loosely supported in Fred Brooks' essay "The Surgical Team."[4] His team of about 10 people includes two who are real experts (the Chief Programmer and the Language Lawyer). Having too many stars creates ego issues and distractions, while not having enough can leave the team struggling with small problems.

The rest of the team should be good, solid developers with compatible personalities and work habits. The more advanced team members can step ahead into uncharted waters, develop the most critical algorithms and applications, and provide technical mentoring to the rest of the team.

The most critical element in selecting people is creating an environment in which they can excel, and that lets you focus more on technology than team dynamics. You don't want a team of clones, but you do want people who are compatible with one another and with the company and team environment you are striving to establish. For example, a married-with-kids, laid-back, nine-to-five developer might not work well on a team of young, single, forceful seven-to-eleven developers. This doesn't mean the former is any less qualified or productive. Actually, that laid-back developer may produce better code and be more productive than the rest of the group. If you think that first person brings a calming, focused influence without either "side" becoming overly frustrated, maybe it is a good fit after all. At any rate, you must take these factors into consideration when building your team.

Wherever possible, and it usually is possible, involve customers and users in the development. Not only does this help build higher levels of trust between developers and users, it also places domain experts within arm's reach of the developers throughout development. This increases the chance that you will develop a product that meets the user requirements.

Give the team what they think they need

Once you have built a strong team, you must next provide it with an environment that encourages productivity and minimizes distractions. First, do your best to arrange quiet, productive office spaces. This is often impossible given most corporate realities, but a comfortable office setting can yield dramatic results. Highly productive environments contain white boards, meeting areas (formal and informal), private office areas, and flexible, modern lab facilities. Add comfort elements such as stereos, light dimmers, coffee machines, and comfortable chairs; you will create an environment where people can focus on their work and forget the rest of the world.

Once you have a team with a productive office space, you need the proper equipment. Do not for any reason scrimp on equipment. The difference between state-of-the-art machines and adequate development systems is less than $1,000. You will probably spend at least $100,000 per year to keep a good developer, including salary, bonuses, benefits, training, and other related expenses. That extra $1,000 amortized over two years represents less

than 1/2 percent of the employee cost.

Finally, your team needs tools. Get good, proven tools from stable companies. Nothing will derail a project faster than using unsupported tools. The team also needs training on those tools; losing files and folders from ignorance and inexperience is painful and costly. The term tool does not just mean compiler. You also need analysis and design, configuration management, testing, back-up management, document production, graphics manipulation, and troubleshooting tools. This is, however, an area where going first-class does not necessarily mean spending the most money. Shop carefully, review a lot of options, and involve the entire team in the decision.

Maintain the Momentum

By now, you have your development team energized with strong co-workers, a great working environment, and some high-end hardware. Congratulations, you have momentum. The next critical factor is maintaining and increasing this momentum. Building momentum initially is easy, but rebuilding it is dreadfully difficult. Momentum changes often during the course of a development effort. These changes add up quickly, so it is crucial to quickly offset the negative shifts with positive ones.

You should focus on three key items to maintain or rebuild team momentum:

- Attrition—keep it low.
- Quality—monitor it early on and establish an expectation of excellence.
- Management—manage the product more than the people.

Attrition

Attrition is a constant problem in the software industry. It can spell disaster for a mid-stream software project, because replacement personnel must quickly get up to speed on software that is not complete, not tested, and probably not well-documented yet. A tremendous amount of knowledge walks out the door with the person leaving, and those left behind have a scapegoat for every problem from then on. Also, in this tight labor market, the lag time between when a person quits and when a replacement is hired can wreak havoc with even the most pessimistic schedules.

Quality

You cannot go back and add quality. By the time you figure out you have a quality problem, it is probably too late to fix it. Establish procedures and expectations for high levels of quality before any other development begins and hire developers proven to develop high-quality code. Have the developers participate in regular peer-level code reviews and external reviews.

Invariably, when a project is cruising along, everybody is excited, the status reports look great, and the GUI is awesome, everything goes wrong. There may be a bad test report, a failed demo, or a small change request from the customer that becomes the pebble that starts an avalanche. You fix one bug, or make one change, and cause two more. Suddenly, the development team that was making fantastic progress is mired in repairing and modifying code that has been in the bank for months.

Management

Manage your product more than your personnel. After all, the product is what you are selling. So, if your corporate culture can handle it, don't worry about dress codes or fixed work hours. Relax and let people deliver things at the last minute. Then critique their products. If the products are not acceptable, you can start working with the individuals to improve their products. The goal here is to not make individual issues team problems. Just because one or two people like to come in at 10:00 a.m. and work until 5:00 p.m., abusing the flexibility you give, doesn't mean you should dampen the environment for the whole team. Too often, project leads avoid confronting the individuals and merely "fix" the problem by setting arbitrary team rules. Soon, everyone is griping about deviant co-workers and the strict management. Those are the sounds of momentum slipping away. Roll a few of these decisions together, and the team is soon focused more on avoiding the rules or tattling on offenders than on producing a quality product.

Project leaders often avoid confronting individuals and merely "fix" a problem by setting arbitrary team rules.

When you do have a legitimate personnel problem, deal with it quickly. If you must let someone go, do it quickly and then meet with your team to explain what happened. As long as you are being

fair, these experiences will contribute to the team's cohesiveness and allow them to rebuild momentum quickly.

Track Progress

Consider the intangible nature of software compared to traditional brick-and-mortar construction. Construction results in a physical manifestation of a conceptual model—the blueprint becomes a building that people can touch and see. They can also touch and see all of the little pieces as they are being nailed, welded, glued, or screwed to the framework during construction. Software development begins as a conceptual model and results in an application, so there is no physical manifestation of software that can be touched and measured, especially during construction.

If you don't take time to figure out what happened during a project, both the good and the bad, you're doomed to repeat it.

A large problem in managing software development is figuring where you are in your schedule. How complete is a module? How long will it take to finish modules X, Y, and Z? These are hard questions to answer, but they must be addressed. If you don't know where you are in relation to the schedule, you cannot adjust and tweak to bring things back on track. Many methodologies exist for tracking progress; select one at the right level of detail for your effort, and use it religiously.

Make Smart Decisions

Making smart decisions often separates successful project leaders from failures. It shouldn't be hard to identify a bad decision before you make it. Choosing to rewrite a few of Microsoft's dynamically linked libraries to accommodate your design choices, for example, is a poor decision. Yet I have seen at least four major projects attempt such insanity. If your application needs to communicate across a serial connection, do you buy a commercial library of communications routines or develop your own from scratch? If you build it from scratch, you can then implement your own personally designed protocol. Bad call. Always use commercial libraries when available, and never try to create a new communications protocol. At best, it will cost you a fortune. At worst, it will sink your project.

People also consistently make bad decisions in selecting technologies. For example, how many people chose to develop applications for the Next platform? Most never finished their applications before that platform went away. When you pick a fundamental technology, whether a database engine, operating system, or communications protocol, you must do a business and a technical analysis. If the technology isn't catching market share and if a healthy company doesn't support it, you are building your project on a sandy foundation.

Because your foresight is fallible, use your design to insulate yourself from the underlying technology. Encapsulate the interface to new or niche technologies as much as possible. Think about which technologies will be prone to change over your product's lifetime and design your application to insulate—to a practical level—your code from those changes.

You will have many opportunities to make good decisions as you negotiate the customer's requirements. Strive to move the requirements from the complicated, "never been done before" category to the "been there, done that" category. Often, users request things that are marginally valuable without understanding the complexity. Explain the ramifications of complicated requirements and requirements changes in terms of cost and schedule. Help them help you.

Post-Mortem Analysis

Few companies institutionalize a process for learning from their mistakes. If you do not take time to figure out what happened during a project, both the good and the bad, you are doomed to repeat it.

What can you learn from a post-mortem analysis? First, you learn why your schedule estimates were off. Compensating for those factors in the next project will dramatically improve your estimating techniques. A post-mortem will also help you develop a profile for how your team and company develop software systems. Most companies and teams have personalities that strongly impact the development cycle. As you go through post-mortem

analyses, these personalities emerge as patterns rather than as isolated incidents. Knowing the patterns allows you to circumvent or at least schedule for them on your next project.

In his book *Managing Software Development Projects*, Neal Whitten offers six steps for executing a post-mortem review of a project:[5]

- Declare the intent: Announce at the beginning of the project that you will hold the review. Also define what topics will be addressed, and set the procedures.
- Select the participants: Choose representatives from each major group associated with the project. To ensure an objective review, management should not participate directly.
- Prepare the review: After the project is complete, assign review participants to gather data. This should include metrics, staffing, inter- and intra-group communications, quality, and process.
- Conduct the review: The actual review should not require more than a few days of meetings. All participants should start by presenting their findings and experiences with the project. Next, the group prepares two lists: things that went right and things that went wrong. Participants can then begin to work on what went wrong to develop solutions.
- Present the results: The participants should present the results to the development team and executive leadership.
- Adopt the recommendations: The company must implement the recommendations on upcoming projects. Without this follow-through, the process yields a marginal benefit.

The premise and benefit of performing post-mortem analyses are validated by the process improvement movement inspired by W. Edwards Deming during the late 1980s and early 1990s. He suggests objectively measuring a given process and using those measurements to evaluate the influence of changes to the processes. Only by measuring a system and analyzing those incremental measurements can you truly improve the system.[6]

Guess what? Your company's software methods and habits for developing software constitute a system. It is far less defined than an assembly line, but it is still a system. The post-mortem analysis allows you to modify that system for the next "production run."

These five critical factors hold true regardless of the design and development methodology, the implementation language, or the application domain. However, this is not an exhaustive list—many other factors influence the successful management of a software development effort. But if you master these five, you greatly increase the odds of completing your project on time and within budget. Just as important, you increase your chances of actually delivering something your users want. ❖

References

1. R. Whiting, "News Front: Development in Disarray," *Software Magazine*, Sept. 1998, p. 20.
2. J. Martin and C. McClure, *Structured Techniques for Computing*, Prentice Hall, Upper Saddle River, N.J., 1988.
3. T. Field, "When BAD Things Happen to GOOD Projects," *CIO*, 15 Oct. 1997, pp. 55-62.
4. F.P. Brooks, Jr., *The Mythical Man-Month: Essays on Software Engineering*, Addison Wesley Longman, Reading, Mass., 1995.
5. N. Whitten, *Managing Software Development Projects*, John Wiley & Sons, New York, 1995.
6. R. Aguayo, *Dr. Deming: The American Who Taught the Japanese About Quality*, Fireside Books, New York, 1990.

About the Author

John S. Reel is the chief technology officer of Trident Data Systems, an information protection and computer networking company. He is a co-inventor of patented COMSEC technology. He received a BS in computer science from the University of Texas at Tyler and a PhD in computer science from Century University. He also worked for the US Department of Defense in systems support, software development, and management.

He is a member of the IEEE, the IEEE Computer Society, Information Systems Security Association, and the Armed Forces Communications and Electronics Association.

Readers may contact Reel at 6615 Gin Road, Marion, Texas 78124; e-mail jreel@tds.com.