# Chapter 3

# The Software Project

The first two chapters were about the software product; this chapter turns our attention to the software project. It introduces you to some powerful mental models for approaching software projects. By understanding and internalizing these models, you can develop good instincts for leading development efforts.

The first two chapters described the software product, its quality attributes and its specification. We now turn our attention to the way in which software is developed. This chapter is an investigation of the software project. All software development projects have certain essential characteristics. Software development can be seen in various perspectives:

- As an exercise in collaborative problem solving
- As a kind of product development
- As nonlinear, dominated by the interactions of the participants

What follows is an explanation of each of these perspectives on software development, along with their implications for leadership. By understanding and appreciating these characteristics, you will have a mental model of software development that will help you make the decisions as if by second nature.

## 3.1 THE DEVELOPMENT PROBLEM

The problem facing the development leader is the creation of quality software within a specified time and budget. We will call this the *development problem*. Recall, in Chapter 1, we defined quality as meeting the stakeholders' needs.

One traditional approach to solving the development problem is to create a detailed project plan. This plan has fixed content, schedule, and budget. Once the plan is developed, all efforts are directed towards meeting the plan. Much of the literature restricts its attention to this solution. In fact, at least one well-known consulting firm defines project success exactly in these terms.

However, this connect-the-dots approach to development is unrealistic. Generally, one does not have enough detailed information to solve the development problem using a fixed, detailed project plan. The requirements are not sufficiently enumerated or even understood, the design is yet to be discovered, the estimates are imprecise. Moreover, as we will see, projects are inherently unpredictable; attempting to hold to a highly detailed plan is futile. A more realistic approach to software development must be found.

We start our exploration of understanding how to solve the development problem by exploring problem solving in general.

Consider how you solve a mathematics word problem. First, you make sure that you understand the problem. You make sure you understand the assumptions and what constitutes a solution. Perhaps you draw a diagram or recast the problem using a formula.

Once you are comfortable that you have a sufficient understanding, you can decide how to approach the problem. You might realize that standard algebra can be applied or that the problem is similar to one you have solved before. As you try to apply your approach, you may realize that you are missing necessary data or that you must go back to enhance your understanding of the problem.

Then, when you are confident that you fully understand the problem and believe your approach will work, you implement the solution. With luck, your approach is viable. However, you may discover that your approach is problematic or may not work at all. In that case, you proceed to refine the approach or find an alternative.

Finally, you verify the implementation by checking to make sure your solution solves the original problem statement. If it does not, you check your work, reconsider your approach, and possibly even rethink your understanding of the problem.

No matter what problem you are solving, from developing a high-energy physics model to installing a home network, you go through the same four phases:

1. Understanding the problem
2. Finding an approach
3. Implementing the approach
4. Verifying the solution

In software development, the entire team needs to solve the development problem collectively. The team as a whole must go through the problem-solving stages. The leader must take his or her team through this process.

## 3.2 DEVELOPING PRODUCTS

*Product development* is the discipline of creating and bringing to market consumer or business products. This field also must deal with a problem that is like the development problem. The product development leadership must find a way to design and bring to market, in a timely and affordable way, a product that meets the stakeholders' needs. Going deeper, we find several similar challenges. The product and software development teams must

- Create an innovative, elegant design that addresses the requirements in some optimal way
- Understand and adopt the latest technology appropriately
- Determine and prioritize user needs while meeting schedule and budget limitations
- Deliver a quality product that will perform well in the field
- Lead a team to achieve a robust, expandable, maintainable design
- Coordinate the activities of multidisciplinary teams
- Integrate technical and marketing strategy

Product development managers are not particularly concerned with generating documents. Their management approach to product development is as a collaborative problem-solving exercise.

In addition, many product developers must worry about manufacturability, generally a small problem for software developers. Therefore, while a few of the details are different, the overall problems are the same. Software leaders can benefit from the lessons learned from the product development community.

### 3.2.1 Product Development Lifecycle

There is no official product development lifecycle. From the various references given at the end of the chapter, you will find the following product development lifecycle is typical. See, for example, Ulrich and Eppinger [2000].

- **Knowledge acquisition**—gaining an initial understanding of the problem
- **Concept development—**creating a design approach to meeting the market opportunity
- **Product engineering**—implementing the concept design and adjusting it as necessary, adding design details to complete a full, detailed specification
- **Pilot production—**building a functional model to verify the solution and addressing any unresolved manufacturability problems

This development lifecycle ends at acceptance by manufacturing. Final bugs may be ironed out during manufacturing ramp-up.

Note how well these phases of the product development lifecycle align with those of problem solving listed in the previous section. During the knowledge acquisition phase, team members do everything necessary to understand what needs to be developed. They may interview users, read marketing reports, and consult with the sales organization. They may meet with management to understand business needs and the marketing strategy. They may develop a competitive analysis and study the underlying technology. The outcome of this phase may be a business case, a product proposal, a vision document, or a requirements specification. In short, the team understands the problem to be solved. When the team thinks it knows enough about the product, it is ready to proceed to developing the conceptual design.

During the concept development phase, the approach to solving the problem is determined. The development team creates a high-level (low-detail) design of the new product. They may draw pictures, develop a virtual prototype on a computer, or create some sort of mock-up. In the auto industry, they build life-sized clay models. An equipment manufacturer may construct a case or cabinet showing dials and displays (the user interface). This mock-up is tested with marketing representatives or a customer group in the user community. Product features are continually re-evaluated, and tradeoffs are made among the market value, the development risk, and the manufacturing cost. During concept development, the team may also consider several alternative design approaches and the tradeoffs among them. The alternatives may be evaluated in terms of the design quality attributes (for example, robustness and maintainability) and other product issues (for example, manufacturability and the cost of the bill of materials). Each design alternative consists of a list of the major product components and how they interact. This phase ends when the team thinks it understands how it plans to shape the design and agrees on the major components.

During the product engineering phase, the team implements the conceptual design. It fleshes out the design details until a full, detailed specification is completed. They may farm out the component designs to different teams. The development team continues to build a series of prototypes that reflect the increasing detail in the design. Team members evaluate the prototypes with respect to quality and manufacturing concerns. For example, two components may not fit together well enough to permit assembly without frequent breakage, or one component may interfere with the replacement of another in the field. Based on these determinations, the component design may be updated, requiring a trip back to the drawing board. This phase ends when the team is comfortable that any remaining design problems can be uncovered only in pilot production.

Eventually, in pilot production a fully functional version of the product is built and ready for transition to manufacturing. In this phase, the team verifies that they have solved the development problem. During manufacturing ramp-up, the development team addresses the operational issues of construction and assembly in the factory setting. They may discover last minute glitches that require some redesign. This phase ends when the product is ready for full production.

Given that the problems faced by software and product development are so similar, it is reasonable to expect that software development should follow phases similar to those of product development. This similarity will be discussed in detail in Chapter 5.

### 3.2.2 Phases and Iterations

One of the salient features of product development is that as the team moves through the phases, much of the team's activity consists of building a series of product iterations. Each *iteration* is a version of the product. As the development goes from phase to phase, the product iterations become increasingly close to final design. In the concept development phase, the iterations serve to help understand the product requirements. They may be mock-ups of the product for market testing. For example, a medical instrument maker may develop a version of the instrument with all of the buttons and knobs and no internal electronics. This mock-up can be used to test out the industrial design with prospective users. Later iterations can be used as a proof-of-concept to test out the practicality of the conceptual design. The final iterations might be used to address reliability and manufacturability issues.

Product development provides some important lessons:

- The maturity of the overall product design is marked by completion of phases rather than by completion of documents.
- Development progress is attained through product design iterations.
- The overall design is tested on an ongoing basis.
- A continual focus on meeting stakeholder needs is maintained throughout all the phases.
- Throughout the process, attention is paid to adjusting the product features to address development and market risks.

To summarize, in product development, the team goes through the phases of problem solving by developing a series of product iterations. As we will see, this approach provides a good model for software development.

## 3.3 SOFTWARE PROJECTS ARE NONLINEAR

Common sense flows from a shared view of how the world works. As shared world views change, so does common sense. Examples of how common sense has changed over recent decades include attitudes toward gender and race, as well as how teams work together.

The old common sense about software development was that the process was linear. The common belief was that teams should be managed like machines, each member doing a highly structured task while working as much as possible in isolation. This approach, sometimes called *scientific management* [Accel-Team.Com, 2000], sprang from the implementation of the assembly line in the early twentieth century and peaked with time-and-motion research in the 1950s. In this mechanistic approach, each person in the business process did one job repeatedly and, presumably, expertly. When their task was complete, they passed the item to the next person in the process. No one worker needed the big picture; they just focused on their own task. Interactions were minimal and the result was exactly the sum of the tasks.

Although an assembly line is efficient at producing many identical items that meet rigorous specification, it was found to have important limitations. First, the mechanistic approach did not provide a means for discovering and addressing certain quality issues. Some defects arose from the coupling of the tasks, from how people work together. For example, suppose one person follows instructions in placing a part on an assembly line. The next person sees that to place his or her part, he or she must hammer it in, degrading the product. If the two workers could collaborate, they might find a way to place the first part so that the second part fits easily. Another problem with scientific management was that people did not like being treated like machines. This management approach resulted in an adversarial relationship with management. The history of labor relations in the auto industry illustrates the point.

Some people see a mechanistic approach to business processes as common sense. Each task must be done. Why not analyze each task in detail so it can be carried out in the most precise and repeatable way possible? The *waterfall* software development process is an example of this approach, and early literature on the waterfall process makes such claims. The problem is that this mechanistic approach makes a leap of faith: that the interaction between team members is as simple as handing off a part in an assembly line.

A consequence of this linear thinking is the belief that the process, not the skills of the individuals, is the dominant contributor to success. The leaders of organizations that adopted this approached invested heavily in corporate processes and their enforcement. Their belief was that if they hired staff to fill the process roles with limited skills, the process itself would magically lead the team to a solution to the development problem. They held to this belief even though the research of Boehm [Boehm, 1981, 2000] and others shows that staff skill is the dominant success factor.

In what follows, we will show that process is important but it is not a substitute for skill. The process must provide a means to enable the skilled staff to work better together. To understand the role of process, you first must understand how members of the team need to interact. Today, we understand that development teams act more like societies or ecosystems than assembly lines. Each member plays a role in the community. These roles, not the artifacts, determine the interactions. With the right kind of leadership, the team members can come together in unexpected, but functional, ways to solve the problem. The dynamics of societies is nonlinear.


The old common sense was about control; the new common sense is about leadership.

### 3.3.1 Nonlinear Dynamic Systems

A system is *dynamic* if, for all practical purposes, it changes over time. A bridge may rust or experience metal fatigue, but it is more or less static. The stock market is dynamic; so is the marketplace. So is your software team: Every day, the people and the work are different.

A dynamic system is *nonlinear* if the response is not proportional to the input; that is, if small changes can lead to large reactions. Machines tend to be approximately linear; most real-life processes, including software development, are nonlinear.

To understand how a nonlinear system operates, consider the following thought experiment:

> An automobile is placed in an empty, uneven field. You are asked to set the steering and speed, and to add the right amount of fuel so that the car, unoccupied, will arrive and stop at a specified target.

On the first try, you point the car in the right direction, compute the necessary fuel, set the speed control, and release the car. The car is buffeted by the wind, affected by the slope and bumps in the field, thrown off track by a pothole, and misses the mark. You try again and again, making all sorts of adjustments, and the car repeatedly misses the target.

You decide to take a systematic approach to the problem by running a series of experiments. You build a data table of initial conditions for the car, including the starting position, the angle of the steering wheel, the amount of gas, and the initial velocity. You set up the car with each of the initial conditions, let it run, and track where it ends. With enough data, you hope to be able to find the right settings to

have the car arrive at the target. You plan several runs with each of the settings and average the results. When you run the experiment, you are amazed to find there is no predictability. The car ends up somewhere wildly different each time, with differences so varied that the averages are not reliable.

Eventually, you realize that unless the field is bowl-shaped and the target is at the bottom, the goal is unachievable. There is no repeatability. You observe that just a little steering would make all the difference. You realize that the outcome is affected by nonlinear interactions: interactions internal to the automobile (suspension, tires, steering), as well as by the outside temperature, the shape of the field, the wind, and many other factors. Each time you run the experiment, these nonlinear interactions affect the outcome. Because the car running in the field involves the nonlinear interaction of all these variables, it is a nonlinear dynamic system.

Scientists and mathematicians have come to understand the nature of such complex systems as the car in the field. An area of study, often called chaos theory [Lewin, 1999; Kauffman, 1996], is used to explain systems that involve many interacting entities. Chaos theory has been applied to weather systems, the dynamics of the marketplace, the origin of life, and, for about a decade, business organizations.

Some fundamental principles describe how nonlinear systems work. They provide guidelines for effective management of complex organizations such as development organizations. Most nonlinear systems are in one of three states, each found in business organizations.

**State 1. Chaotic: Unpredictable and unadaptable**

The behavior of chaotic organizations is always changing. The workers' tasks change frequently, so no one is sure who is doing what or who is responsible for what. People come to work each day trying to find a way to move the project forward. Attempts to get things moving fail. Interactions generate friction, and there is no discernable progress.

In disorderly projects, under chaotic conditions, team members do not communicate. They are continually in each other's way, unsure of what their job is. They spend more time trying to coordinate their efforts than moving the project forward. This kind of internal friction generates heat and not light.

Staff members of chaotic projects are emotional in interviews. They usually complain that what they do does not meet their job description, that they do not get credit for their efforts, that their time is wasted in unproductive meetings, and that their management does not seem to have a clue about what is really going on.

**State 2. Stable equilibrium: Predictable, but unadaptable**

Systems in a stable equilibrium behave as the car would if the field were a bowl. No matter where you set the car in the field, the car winds up at the bottom, in a state of equilibrium. The outcome is always the same. These systems require a large amount of external energy to move them from equilibrium. To carry the metaphor further, moving them from their stable state for the long term is like carrying the car out of the bowl. Small changes will not achieve this result.

Some managers believe that a stable equilibrium is good for their organizations. They perform year after year, seemingly impervious to the world around them. Their groups are not only easy to manage; they are hard to damage, even by poor management.

Sounds good, but there is a serious downside to the equilibrium state. Such organizations are very difficult to change, like the car at the bottom of the bowl. When the pressure to change is on, they resist new behavior; when the pressure is off, they revert to old behavior. These organizations do not compete effectively when changes in mission or technology are necessary. There are many examples, such as organizations comfortable with 1960s mainframe-based development methods. Given the amount of change in our field, these stable organizations are doomed. However, organizations in stable equilibrium are the exception.

**State 3. Edge of chaos: Unpredictable, but adaptable**

For most real systems, small changes in initial states can result in very different results. No amount of precision and detail is enough to set these systems on a course to a predictable end. Unpredictable behavior is intrinsic to these systems and cannot be overcome.

There is a state that is not in equilibrium and not fully chaotic. When systems are in this state, without steering it is impossible to predict their future state from knowledge of the present, like the car in the uneven field. However, these unpredictable systems do respond to external influences, so that they can be influenced to achieve a useful purpose. In the automobile example, a little steering makes it easy to hit the mark.

Edge-of-chaos systems have several remarkable properties that form a basis for understanding how best to organize and lead complex enterprises such as software development organizations. The first property is that small changes have large effects. As with the car experiment, a small change in steering direction makes a big change in the car's eventual position. This property is a defining char-

acteristic of edge-of-chaos systems. It follows that it is impossible to predict exactly how these systems will respond to a plan or a set of directives. In fact, the greater the detail, the greater the unpredictability.

Fortunately, these systems are manageable, if not controllable, as a result of perhaps their most remarkable property: They spontaneously organize themselves into apparently orderly systems that adapt to changes. However, for this to happen, the system must consist of independent entities with the freedom to interact. With this freedom, the entities create the necessary connections and communications paths. System order does not arise from being controlled, but instead by being *managed*. In fact, any attempt to apply control may result in the system becoming locked in a rigid equilibrium structure that cannot make the necessary connections to deal with a changing environment.

## 3.4 TEAMS AS DYNAMIC NONLINEAR SYSTEMS

Development teams are made up of interacting individuals with different perspectives who generally do what needs to be done to complete a project. There are designers and testers, people who understand the application technology, and experts in software technology. Each is important, with unique experience, perspective, and priorities; each has a role to play. Throughout the project they need to interact, to communicate, to raise and address issues. As the project proceeds, the nature of the interactions changes. Each individual's interactions combine to become the team behavior. The theory of dynamic systems applies to this collective behavior of a team performing a development project.

Individuals typically fill one or more roles on a development team.

- The project manager, or lead, develops and maintains the project plan, including cost and budget; prioritizes and schedules the content; staffs the project; and provides day-to-day leadership.
- The project architect looks after the overall design and associated quality issues. Responsibilities include maintaining the integrity of the UML views, especially the top-level logical decomposition, and the problem report database. Brooks [Brooks, 1995] and others (e.g., [Booch, 1995]) consider the architect a key team member.

- The business modeler models the business process that the system under development is intended to support.
- The project requirements analyst analyzes the business model and other user input to develop the requirements database, including use cases and supplemental requirements.
- The developer designs, implements, and tests individual object classes.
- The integrator or configuration management staff defines the project code library structure, instantiates the configuration management environment, designs and implements the method of compiling and integrating the code (the "make" files), and creates the builds for testing and delivery.
- The tester develops test plans, carries out system and component testing, and generates test and problem reports.
- The documentation writer creates the help file and the content of the user guide.

Large projects require partitioning into several development teams, with each team responsible for some number of logical subsystems. With the exception of the documentation writer, each role should be instantiated within each team and at the project level. For example, there should be a project architect and a team architect on each team. The project architect looks after the overall system design; a team architect looks after the design of the assigned subsystems. All of the architects should form a joint design team. This sort of structure enables the communications necessary to achieve a coherent design. The same rationale applies to the test staff and the project team managers.

The person in each role has primary responsibility for one or more development items. However, they cannot work in isolation; they must communicate with other team members to be effective.

### 3.4.1 Order and Team Communications

One distinguishing characteristic of a development project, as a dynamic system, is that the entities are people, neither machines nor chemical molecules. The emotions and motivations of the team members are among the conditions and variables that interact to govern their behavior. Applying leadership, you affect these variables and steer the project.

Several scientists have considered models of how systems of interacting discrete entities evolve [Kauffman, 1996; Lewin and Regine, 2000]. They have found that the nature of the system is determined by communications among the entities:

- If communications are too restricted, the system will be in a stable equilibrium.
- If communications among the entities are excessive, the system becomes chaotic.
- A middle range is optimal. With just enough communications, the system is at the edge of chaos and evolves stable, manageable structures.

The following observations about communications are directly applicable to project teams, which consist of independent, interacting entities.

- When a team is too restricted in its communications, it does not have the means to adjust to the challenges of a development project. This can happen if the manager insists on being in all of the communication paths.
- If communications are unstructured, the project is likely to become chaotic. This can happen if the manager is uninvolved and the team must organize itself.
- One of the critical management tasks is to develop an organization that enables and promotes the right amount of communications.

This observation reinforces the wisdom of Brooks [Brooks, 1995] and others:


The key to leading successful projects is the right amount of communications.

This idea is entirely consistent with the software development economics model discussed in Chapter 4.

This new common sense provides a final insight: The global behavior of a nonlinear interacting system emerges from the totality of local interactions. It follows that you cannot mandate how your team will behave, but you can influence its behavior by the nature of your communications with team members. Think of this kind of communications as constructive involvement. Being a part of the team

allows you to provide the necessary steering to keep the team focused on delivering the right product and to influence overall team performance.

Chapter 6 discusses some mechanisms for enabling the correct amount of communications and participating constructively in team efforts.

## 3.5 THE PROJECT PLAN

The automobile thought experiment is applicable in several ways to software development. In the thought experiment, the goal is to have the car arrive on a target; in software development, the goal is to have your team deliver a solution to the development problem. The initial conditions in the auto experiment are the setting of the steering wheel and the amount of gas in the tank; in the development problem, the initial conditions are the staff on board and their understanding of the requirements.

Note that a classic project plan pictured in project management books and supported by project planning tools is a linear object. It breaks the work into small pieces and adds them together to determine the effort and schedule. It does not account for the nonlinear nature of the interactions among the team members. As linear objects, these project plans are only approximations of what will take place. The plans then are useful only if they are not taken too seriously. As with many linear objects, a project plan can serve as a useful approximation.

Project plans are also predictions of the future. They contain information on what each of the staff members will be doing when solving the problem. Insights from chaos theory tell us that the information contained in a real-life project plan is unknowable. Insisting that a team hold to a frozen plan ignores the fundamental nonlinearity of the process. In fact, you can be sure that the initial plan is almost certainly not accurate. This does not mean you should not have a plan, but only that you must refine and update the plan continually throughout the development.

This process of refinement and updating enables the plan to serve as a steering mechanism. This requires the plan's details to be filled in as the project evolves. The team's experience in a given phase serves as input to the details of the plan for the next phase. This iterative development of the project plan is one of the features of the Rational Unified Process, introduced in Chapter 5.

### 3.5.1 Less Is More

Some managers and staff process engineers do not understand the nonlinear nature of their projects. They believe that the reason project plans aren't followed

is that they are not sufficiently detailed. They believe that adding granularity to a project plan creates order. (Adding *granularity* means adding tasks of a shorter duration.) For example, the manager may respond to an identified shortcoming in a previous project by adding more work items. Of course, more detail only makes things worse. Adding details takes the leap of faith that one can predict the future with great fidelity.

I have seen six-month projects with 1,800 work items. Based on 180 working hours in a month, each item, on average, would be less than two hours in duration. Anyone trying to administer such a plan would spend considerable time just updating status. Each developer from hour to hour would have to report which of the work items was being addressed. Fortunately, most organizations wisely ignore managers or process groups that try to impose such plans. The downside is that the projects are left with no plan at all. Ironically, the attempt to impose order results in chaos.

## 3.6 APPROACHING DEVELOPMENT RISK

One way that people think about development projects is an exercise in risk management. Some consultants on risk management recommend that the team identify all of the possible risks, usually through some sort of brainstorming session, and then ensure the risks are mitigated. While understanding and addressing the risks of your project is important, it is best to have a more structured way of thinking about risks in your projects.

People can only think about seven things at a time [Miller, 1956]. Given the complexity of software development, it is important to focus on the right things. A useful way to identify and track the issues in managing a software project is to track a small number of development risks.

The literature on software risk analysis is extensive and typically overly elaborate: lists of hundreds of possible risks; bureaucratic procedures that consist of risk management plans, brainstorming sessions with risk identification, and periodic assessments. Most of this is wasted effort. In the spirit of keeping things simple, you should focus on the three project risks:

- Schedule risk: Not delivering the project on time
- Cost risk: Exceeding the budget before you deliver the project
- Quality risk: Delivering software that fails to meet stakeholder needs

Other so-called risk items such as staffing risk (the inability to staff) or technical risk (uncertainty about how to design the code) put the project at risk only to the extent that they affect schedule, cost, or quality. For example, the inability to staff is only a problem if it will put the schedule at risk.

### 3.6.1 Schedule Risk

The product development perspective addresses schedule risk in three ways:

1. It provides insight into the progress of the software development. By tracking the progress of the actual product and not trying to gauge completion of the activity, the manager can determine the true schedule variances. This accurate and timely insight enables the manager to address schedule risk.
2. It provides ongoing integrations and prototypes. Because this approach supports evolving versions of the software rather than integration at the end, technical risk is distributed throughout the effort rather than being stacked at the end.
3. It provides phases with planned milestones. Planned milestones provide momentum for completion of the effort. The defined phases with specified completion criteria allow the manager to keep the team focused on the dates. The phases help keep the project from spiraling out of control.

### 3.6.2 Cost Risk

The attributes of the product development perspective that help address schedule risk also apply to cost risk. In addition, the manager can prioritize content throughout the development. Iterations are planned so that the most essential features are addressed early. As development progresses, the manager can drop unessential features as necessary to hold to the budget, based on cost/benefit tradeoffs. In contrast, the systems engineering approach does not provide the mechanisms for making such tradeoffs once the specifications are written.

### 3.6.3 Quality Risk

The product development method addresses quality risk in several ways. The underlying approach is to find a solution to the development that satisfies the

product needs. The manager can keep the team focused on the system architecture as the solution evolves. Finally, with good schedule and cost management, the team will not be scrambling to patch together a solution at the end of the project. They will have time to address the quality attributes.

The manager can review the software with stakeholders at each iteration of the evolving software. Necessary changes and tradeoffs can be made throughout the project. The product development approach does not rely on perfect knowledge and understanding at the beginning of the project.

## 3.7 A WORD OF CAUTION: NO SILVER BULLETS

The product approach may be a good framework for understanding system development, but it is not a silver bullet. Its application takes a significant investment from software managers: You must understand how to apply the approach in a detailed, disciplined way and lead your organization in adopting it. Fortunately, there is a detailed software methodology, the Unified Process, that provides the activities, phase specifications, and milestones needed to apply the product development approach to software. The Unified Process is discussed in Chapter 5.

There is a second type of investment for software managers: The success of the product development approach requires the ongoing involvement of all levels of management. This is a good thing. It is addressed in Chapter 6.

### To Learn More

Here are some excellent references on product development. I most highly recommend the first reference.


- Ulrich, Karl and Steven Eppinger. *Product Design and Development*, McGraw Hill, 2000.
- Wheelwright, Steven C. and Kim B. Clark. *Revolutionizing Product Development*, The Free Press, 1992.
- Tabrizi, Behnam and Rick Walleigh. "Defining Next-Generation Products: An Inside Look," *Harvard Business Review*, November–December, 1997.
- Gorchels, Linda. *The Product Manager's Handbook*, NTC Business Books, 1995.

For over 20 years, Prof. Barry Boehm and his students at the University of Southern California have been studying software development productivity. The results of this effort may be found in

- Boehm, Barry. *Software Engineering Economics*, Prentice Hall, 1981.
- Boehm et al. *Software Cost Estimation with COCOMO II*, Prentice Hall, 2000.

Many who develop software believe that the chaotic nature of the field is due to its immaturity. These texts show the dynamics are much the same for more established fields.

- Sabbaugh, Karl. *A Jet for the New Century: The Making and Marketing of the Boeing 377*, Scribner, 1996.
- Sabbaugh, Karl. *Skyscraper: The Making of a Building*, Penguin, 1991.

Fred Brooks was the first author to document the nonlinear nature of software projects in this classic:

- Brooks, Frederick P., Jr. *The Mythical Man-Month* (Anniversary ed.), Addison-Wesley, 1995.

This is one of the first books to explore the management implications of object-oriented development. It has many good insights.

- Booch, Grady. *Object Solutions: Managing the Object-Oriented Project*, Addison-Wesley, 1996.

There have been several books for the layman on the nature of nonlinear systems. Three of my favorites are

- Bak, Per. *How Nature Works*, Copernicus, 1999.
- Kauffman, Stuart. *At Home in the Universe: The Search for Laws of Self Organization and Complexity,* Oxford University Press, 1996.
- Lewin, Roger. *Complexity: Life at the Edge of Chaos* (2nd ed.), University of Chicago Press, 1999.

One of the best books on the application of nonlinear dynamics to management is this recent text.

- Lewin, Roger and Birute Regine. *The Soul at Work: Listen, Respond, Let Go,* Simon & Schuster, 2000.

This is the classic paper on the limits of people's ability to process information. You will be able to find it online as well.

- Miller, George A. "The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information," *Psychological Review, 63*, 81–97, 1956.

At this writing, a summary of the history of scientific management may be found at

- Accel-Team.Com, Scientific Management. Lessons from Ancient History through the Industrial Revolution, online, http://www.accel-team.com/scientific/index.html, 2000.