Part 1

Getting Started

1

Think Like an Engineer—Especially for Software

Software engineering as a discipline needs a sturdy underpinning of classic engineering principles and discipline. Societies depend on engineers to keep bridges from falling down, factories from exploding, and generally to protect us from our carelessness and ignorance. Software now drives much of our lives, so computer scientists need to accept the engineer's responsibility to produce something that will work reliably and protect human life and work. Engineers do pretty well now with the nuts and bolts of daily life, but that wisdom did not come cheaply. It was built up from thousands of individual experiences of success and analyses of failures, codified, and passed on to new people in the profession. Computer scientists can learn to be engineers in the same way, although our materials are less tangible and our constructs and stresses are measured in different ways.

It is possible to experience the principles and theories of quantitative software engineering in a controlled environment before applying them in a live business project. "Quantitative" is the operative word. Software engineering practices are designed to make the development of software less chaotic, reliably repeatable, and more humane, but unless there are specific measurements to apply, volumes of good practices would make better doorstops. Extraordinary people who are highly motivated can make any process work, but dream teams are rare and burning out talent is shortsighted, cruel, and expensive. Heroics in software development are an indication of process failure that leads to dysfunctional behavior in both organizations and

Trustworthy Systems Through Quantitative Software Engineering,
by Lawrence Bernstein and C. M. Yuhas

individuals.[1] Worst of all, the resulting projects are difficult to maintain, difficult to upgrade, and the developers have little learning to bring to the next project.

We will use a model-based approach to software development. Models will be used to calibrate, bound, and validate your estimates. There will be cost-estimation tools, risk definition and analysis, and prototyping. System models and scenarios will produce test results with actual system performance. You will find problems and solutions, case studies, and "magic numbers" in each chapter. The latter are easy-to-remember rules gleaned from the experiences of people who have earned the right to call themselves engineers of software or from explanations of folklore whose meanings are buried in the mists of early computing.

1.1 MAKING A JUDGMENT

Engineering is a balancing act. When applied to software, the spinning plates are functions provided, time to produce, cost, and complexity. The fundamental software reliability equation is as follows:

$$\text{Reliability} = e^{-k\lambda t},$$

where k is a normalizing constant, λ is complexity/ effectiveness $\times$ staffing, and t is the time the software executes from its launch.

This model equation only approximates reality, but it is useful for making engineering tradeoffs if it is stipulated that software fails at a constant rate. It is reasonable, if unorthodox, to model the software engineering *process* based on this model. Field failure rates for IBM and Microsoft products show a constant failure rate 10 months after product release.[2]

The complexity factor incorporates the elements the software project manager controls through the development process. The software engineer might alter effectiveness by providing better software tools, such as higher level languages, to designers and thereby increase the reliability of the final product. By reusing reliable components, the software engineer reduces the complexity of the system, which again makes it more reliable. By adding staff beyond the minimum predicted by staffing models, more effort can be placed on such activities as diabolic testing and system audits in the interests of improving reliability.

[1] Yourdon, Edward. *Death March: The Complete Software Developer's Guide to Surviving "Mission Impossible" Projects*, Prentice Hall, Englewood Cliffs, NJ, 1997.

[2] Chillarege, Ram, et al. "Reflections on Industry Trends and Experimental Research in Dependability," *IEEE Transactions on Dependable and Secure Computing*, IEEE Computer Society, Vol. 1, No. 2, April–June 2004, Figure 5, www.computer.org.

The reliability equation is the framework for quantitative analysis and making tradeoffs. The software project manager must invest in these ongoing activities: measuring complexity, measuring effectiveness through investments in tools and technology, and measuring staffing requirements.

On your way through several approaches to software design, you will acquire a background in why systems fail and how to avoid failure with risk containment techniques. Topics include risk identification and analysis, designing for reliability, design simplification, and testability. The merits of top-down and bottom-up design are compared. Code reviews and inspections are used with static quality assessment techniques. Testing approaches include unit, integration, stress, reliability, and diabolic testing. Rapid prototyping, top-down, bottom-up, successive refinement, extreme programming, design constraints, and data abstraction are presented. You, the quantitative software engineer, are responsible for producing studies for each activity.

Optional organizational structures can help with ways to manage suppliers, determine span of control, identify key employees and retain them, examine staff churn, and conduct performance evaluations. The economic drivers of diminished defect leakage, earned value, and economic value-added can help to evaluate project viability. The capability maturity model for software (SW-CMM) was created in the early 1990s by the Software Engineering Institute at Carnegie Mellon University to gauge the sophistication and reliability of software products coming out of any given organization. When applied to an organization's methods of developing software, SW-CMM assigned a level of sophistication and reliability to the process and product ranging from a low of 1, initial; through 2, repeatable; 3, defined; 4, managed; to a high achieved at the time only by NASA of 5, optimized. Most organizations were rated only a 1. For most, level 3 was a sufficient goal.

In 2000, SW-CMM was upgraded to the capability maturity model integration (CMMI). CMMI is now being adopted worldwide, including North America, Europe, India, Australia, Asia Pacific, and the Far East to provide models for process improvement.

The CMMI best practices help organizations to more explicitly link management and engineering activities to business objectives. The product or service is the center of engineering activities to make sure it meets customer expectations. CMMI also encourages robust, high-maturity practices in areas such as measurement, risk management, and supplier management. Businesses are helped to comply more fully with relevant International Standards Organization (ISO) standards.

Because this is a quantitative book, occasionally there will be examples with solutions in addition to problems at the end of each chapter. Keeping in mind that, as Hamming remarked, "The purpose of computing is insight, not numbers,"[3] consider the ramifications of the following example.

[3] Hamming, Richard. *Numerical Methods for Scientists and Engineers*, McGraw-Hill, New York, 1962.

EXAMPLE:

A software system is designed so that after every hour of normal operation, it stops and relaunches from its initial state. This process is called software rejuvenation. Assume that the mean time to failure for the software is 10 hours and the mean time to repair is 5 minutes. Repair means restoring the software to an operating condition by relaunching it from its initial state.

What is the probability that the system fails within 30 minutes of operation? Note that e^x is approximately $(1 + x)$ for small x.

SOLUTION:

$R(30) = 1/e^{\{30/(10 \times 60)\}} = e^{-0.05}$ using the approximation, $e^x \cong 1 + x$ for small x, we get $R(30) = 0.95$, probability of failure $= 1 - R(30) = 0.05$. This means that there is a 5% chance that the software system will fail in its first 30 minutes of operation. Without doing the quantitative analysis, the software engineer would not understand the risks of releasing such buggy software.

1.2 THE SOFTWARE ENGINEER'S RESPONSIBILITIES

The software engineer uses critical judgment and analysis to make informed decisions. These decisions typically involve making engineering tradeoffs, but this analysis is meaningful only when supported by project data. You will learn a framework for exercising informed engineering on software products. Skilled software project managers produce systems that meet customers' needs within budget and on schedule. This framework captures, in a quantitative way, the thought processes of skilled managers. This approach uniquely weaves software engineering theory and case histories, quantitative analysis, and technology into the project effort.

This textbook models industrial software development. Teamwork and cooperation are encouraged. Teamwork is one of the hardest lessons for students because they have been schooled in competing for so long. Quantitative analysis is required. Knowledge of, and sensitivity to, software engineering ethics are stressed.

1.3 ETHICS

The ethical software engineer makes sure that a product solves the customer's problem, that it is tested, that good software engineering practices are used in its development, and that any limitations of the product are clearly stated.

Professions are defined by the willingness of their practioners to establish and abide by a code of ethics. Human nature being variable as it is, meaning-

ful disciplinary action is necessary to make ethics stick. There is currently no penalty for software engineers who are unethical, but new laws may change this. What follows is the ratified ACM/IEEE-CS Joint Task Force on Software Engineering Ethics and Professional Practices (Reprinted by permission). It defines the minimum behavior one must exhibit to be truly professional.

ACM/IEEE Software Engineering Code of Ethics and Professional Practice (Short Version)

PREAMBLE

The short version of the code summarizes aspirations at a high level of the abstraction; the clauses that are included in the full version give examples and details of how these aspirations change the way we act as software engineering professionals. Without the aspirations, the details can become legalistic and tedious; without the details, the aspirations can become high sounding but empty; together, the aspirations and the details form a cohesive code.

Software engineers shall commit themselves to making the analysis, specification, design, development, testing and maintenance of software a beneficial and respected profession. In accordance with their commitment to the health, safety and welfare of the public, software engineers shall adhere to the following Eight Principles:

1. PUBLIC—Software engineers shall act consistently with the public interest.

2. CLIENT AND EMPLOYER—Software engineers shall act in a manner that is in the best interests of their client and employer consistent with the public interest.

3. PRODUCT—Software engineers shall ensure that their products and related modifications meet the highest professional standards possible.

4. JUDGMENT—Software engineers shall maintain integrity and independence in their professional judgment.

5. MANAGEMENT—Software engineering managers and leaders shall subscribe to and promote an ethical approach to the management of software development and maintenance.

6. PROFESSION—Software engineers shall advance the integrity and reputation of the profession consistent with the public interest.

7. COLLEAGUES—Software engineers shall be fair to and supportive of their colleagues.

8. SELF—Software engineers shall participate in lifelong learning regarding the practice of their profession and shall promote an ethical approach to the practice of the profession.

These goals are exemplary, but in the rough and tumble of business, they can be overruled in the interests of expediency. Customers, as half of the partnership, must insist that every product have a named software architect and

software project manager to assign responsibility specifically and to have a firm point of control. The same person may perform both roles.

The software architect affirms that the software product solves the customer's problem; affirms that the software product is suitably reliable, easy-to-use, extendible, not harmful, and robust; and affirms that the requirements are valid.

The software project manager affirms that the software was successfully tested against the requirements; affirms and identifies that good software engineering processes were used in the software development and integration; and affirms that the project is within budget, on time, and performs satisfactorily.

Case Study[4]: The Case of the Sacrosanct Date

You are a successful project manager. The boss of the boss of your boss wants you to assume the management of a project in deep trouble. It is a mission-critical, complicated store-and-forward message switching system requiring a large database and significant communication software for computer-to-computer interfaces. The following issues have confounded the current project manager:

- The software is fragile. Its mean-time-to-failure is 2 hours, measured by field-reported crashes.
- This is a large project with 100 people. There are 30 developers and 10 testers. There are five human factors designers helping the users cope with the system deficiencies. They cannot alter the design, to which they had little input.
- The next release of the software is scheduled next week. This schedule has been in place for 1 year, and the customer purchased the system with the assurance that the feature package in this upgrade would not be delayed. The schedule is now in jeopardy.
- Release testing is going poorly; developers and testers are often diverted to find and fix field problems.

Question: Do you accept the position?

Answer: Only if you have the authority to fix the situation. The best time to have a clear understanding of job expectations is when you agree to take on new responsibilities. You need to understand your authority, the resources available, and the consequences of failure. Responsibility without authority leads to frustrations.

Question: You cannot resist a challenge and take the job. Now what do you do?

[4] Thanks to Associate Professor A. David Klappholz of the Stevens Institute of Technology for helping to write several of these case studies.

Answer: Delay the release and stabilize the software (debug it!). Talk directly to the customer and say you need time to make the system stable because you will not ship faulty software just to meet a schedule.

Question: The customer wants a special utility that recovers the system after a crash. To implement such a utility, you need 4 months and two of your best developers. So what do you do?

Answer: Explain that it is better to fix the system so that it does not crash in the first place. Tell the customer that he will get his recovery utility, maybe in 8 months, after you do a quantitative analysis of the resources required for all the tasks facing you.

Question: The customer is angry. At a follow-on meeting with you and your boss, the customer says, "I do not want to do business with your company if you can't see that that recovery utility is my highest priority." Your boss wants to keep the customer happy. What is the ethical thing to do?

Answer: Explain the situation to your team. Focus the team on fixing the stability problems, plan for the utility, and keep your boss, the customer, and the customer's people aware of the steps you are taking.

Question: The crisis is under control so far; however, the pressure from the customer continues. The customer uses the progress data you supplied against you. Your reputation is at stake. What do you do?

Answer: Continue sending progress data to the customer and your boss. Review the situation with your boss's boss to make sure you still have internal support. Work to establish your credibility with your team; otherwise, they might attempt heroics to placate the customer. If that happens, they will not be working to stabilize the software, and you will lose control of the project.

Question: How do you get the team's compliance?

Answer: IT professionals already have high salaries and benefits, so these things will not motivate them. Fear of job loss will keep some loyal, but not necessarily the best. You must keep the team well informed of the steps you are taking and your reasons for taking them. You might invest in a small study for new technology. Giving the team the chance to advance their skills excites them and builds loyalty to your stabilize-then-add-features plan.

Question: Some developers on the team are happy with the current situation. They like the thrill of solving problems and being a hero. But every time they put in a fix, several other things go wrong. How do you deal with them?

Answer: Explain the reality of this magic number at your next group meeting. Post copies of it everywhere. Stop rewarding "heroic" behavior by turning to these people in a crisis.

MAGIC NUMBER!

It takes ten times the effort and money to find and fix a problem in the test and integration laboratory than it would have taken the developer to fix it during the unit test. If the bug gets out the door and the customer has to report it, the fix costs 30 times what it would have taken the developer to fix it originally. Field fix : developer fix::30 : 1!

Then give the testers the "right of rejection." Explain that their job is to assure stability and quality and that it is the developers' job to fix problems on time. Insist that testers reject software that testers judge to be unreliable.

Question: Before you took charge, your team shipped blank tapes on the previous release date to buy time. They reasoned that the customer would take a week to install the software in their test site and that integration testing could continue. The entire release would then be sent as a large fix. Based on this unethical behavior, the customer, quite rightly, does not believe that you will deliver on your schedule commitments. How do you restore that confidence?

Answer: Establish regular meetings with the customer to show progress. Make it clear that the persistent addition of features creates an atmosphere that breeds desperate measures. Therefore, the only person who can commit to additional features is you.

Question: People are afraid of talking about the problems in the software because they fear that, as the bearer of bad news, they will get into trouble. How do you get these people to speak?

Answer: Celebrate the courage and perceptiveness of people who find problems. Each discovered problem is a problem that the customer will not have to face.

Conclusion: Make sure you stabilize the software, deliver it on the revised schedule, and deliver the utility as promised. Divide and conquer by forming

a crisis team and a next-release team. Commit based on this two-team approach. Set goals that exceed the commitments, leaving the difference between the customer commitments and your team goals as slack that you, as project manger, can use to handle unforeseen situations.

1.4 SOFTWARE DEVELOPMENT PROCESSES

There is no best approach to software development. As is true for all engineering disciplines, the approach used and the tools chosen depend on the problem, the skill of the engineer, and the money available. What is the best choice? **It depends!** Quantitative software engineering is aimed at gathering data so that each choice can be insightful.

New development processes evolve as problems become more complex and engineers become more adept at handling software. One of the first development processes defined was the Waterfall Model. Current processes are the agile method, including commercial off-the-shelf (COTS) based and open source-based development, but new and better methods are always being created to adapt to new developments.

Bitter experience with building software that did not do what the client wanted inspired the creation of methods for requirements development. These methods examine ways to distinguish vital requirements from the merely important and to tease out hidden requirements. Descriptively stated requirements must be translated into firm specifications that include an expected operating point and the expected range of performance.

EXAMPLE:

A descriptive requirement might be, "The Customer Resource Management System must respond to online customer Web inquiries for account status." What can a designer do with this requirement?

SOLUTION:

It must become a specification before it can be a design objective. Let us take just one element of this descriptive requirement—respond. In what way? If there are errors, how shall they be described? Suppose the customer needs help? How much of the account should be shown? How fast must something appear after the customer presses ENTER?

Consider only the issue of speed of response. Jones remarks that, "The meanings of various lengths of elapsed time do not vary widely from one person to another: Less than 1/3 sec is 'instantaneous', Less than 1 sec is 'fast'. Less than 5 sec is a 'pause' and Greater than 10 sec is a 'wait.' Transaction inter-

actions should be without 'wait.'"[5] Today, human factors research describes three levels of human cognitive experience.[6] "Perceptual processing time (about 0.1 second) is the time the human perceptual system spends integrating and processing signals. Two stimuli within this time seem fused, and responses feel instantaneous. . . . Immediate response time (about 1 second) is the smallest time needed to react to a new situation—for example, the appearance of a new form on a screen. Unit task time (about 10 seconds) is the time scale of simple tasks."[7]

This bit of data gathering has often been translated into a 3-second response time specification. Other studies show that variability of the response time dissatisfies users more than an average response time. A good designer seeks to relax the 3-second operational specification to 6 seconds with a 1-second bound. Most companies would accept this compromise. This discussion is just one part of one simple requirement.

"Respond" to the mind of the software developer means the execution time needed to retrieve and format the information inside the computer. A hidden requirement is that the communications link to the data server must support the speeds of data transfer needed. A hardware engineer would check to make sure that the data links have enough capacity. But whose responsibility is it to make sure that the communications software within the data server is configured properly and that the software communication drivers can support the transmission links, the protocols, and provide any needed buffering?

Too many developers are unaware of the need to perform this engineering function. Without a good job here, exhausting buffer pools could lead to intermittent delays in the response time to the customer. A prototype is the best way to validate specifications and to let the customer understand the interface before major investments are made in implementation.

1.5 CHOOSING A PROCESS

The software project manager must choose the process the team will follow to produce the software product. The choices range from the top-down, document-driven Waterfall Model and similar planned methods to flexible and responsive agile methods. Each process evolved out of whatever process was in use at the time in response to a core element that was deemed critical, but missing. At various times, those core elements have been documentation, schedule, functions, risk analysis, or hierarchical control, but no one process was meant to totally replace what came before. As each process evolved, layers

[5] Jones, T. Capers. *Four Principles of Man Computer Dialog Computer Aided Design*, Vol. 10, No. 3, May 1978, p. 197.

[6] Newell, A. *Unified Theories of Cognition*, Harvard University Press, Cambridge, MA, 1990.

[7] Obrenovic, Zeljko and Stancevic, Duse. "Modeling Multimodal Human-Computer Interaction," *IEEE Computer*, Vol. 37, No. 9, Sept. 2004, p. 67.

have been added, emphases have shifted, and the result is a continually improving, responsive mixture of tools. Sometimes the business constraints will direct the choice of method and sometimes the project manager's personal style will dictate the most comfortable choice. There have been analyses of management styles that favor structure, control, and authority versus those that favor team-building, collaboration, and flexibility, variously called Types A and B or Theories X, Y, and Z[8], but ultimately each process will emphasize one core element and include the others to a lesser extent. Proponents tend to argue for their own (most comfortable) approach, but it is best to fit the solution to the problem.

The only true failure is to create a software product without any method at all. Novices will want to restrict the size of the product to what three or four people can do, create a set of features as they go along, and then deliver a working system to a customer. Typically they have special domain knowledge that is built into the product and delights the customer. They have found what they think is the silver bullet of software development (Fred Brooks to the contrary!). When they try to build a follow-on system, they find that it is bigger than the tight-knit team can handle, key members of the team have moved on, or their tool base changes. Then they must adapt. Unfortunately, the more technically competent a team is, the more resistance they have to new technology. The organizational device of an "Office of Technology Planning" helps organizations adopt technology and keep current.

When teams are successful, they are asked to build other systems or to extend the first system. As the number of customers grows, they are asked to add customer-peculiar features. These features may be incompatible with other features of the original system, so the number of system versions grows. There are no economies of scale. To handle a growing business, the only thing most entrepreneurs can think to do is hire more people like themselves who also have no knowledge of process, configuration control, or release control. When a customer calls with a serious problem, like total system failure, there is a mad scramble to reproduce the problem. First the developers must reproduce the configuration the customer is using. If they do not know it, a designer is quickly dispatched to the customer site to fix the problem. Without change control, the problem is not recorded and other customers may report similar problems. This Keystone Cops routine was true in the 1960s and still occurs today.

There are too many design and development methodologies to discuss them all. The particular method a project adopts should provide some structure and discipline while being compatible with the abilities of the designers and developers. The best methods tighten the structure and discipline as the project proceeds through the development cycle. Loose control at first encourages several cycles of design synthesis and analysis before the developers commit to

[8] Leavitt, Nancy and Bahrami, Homa. *Managerial Psychology*, University of Chicago Press, Chicago, IL, 1988. Peters and Waterman. *A Passion for Excellence*, Random House, New York, 1985. Ouchi. *Theory Z*, McGraw-Hill, New York, 1981.

production code. Table 1.1 shows some methods. Simplicity in design is an important element of successful systems development, whatever the process.

TABLE 1.1. Popular Software Development Processes

Method	Description
Evolutionary	Software requirements and design will change and grow throughout the development process. Often associated with user-oriented systems or systems not yet fully understood—high volatility.
Incremental	A linear model of the software development process that allows the software developer to iterate among the activities within each life-cycle phase for each increment defined for the system.
Object Oriented	The use of all object-oriented techniques for requirements analysis, design, coding, and testing by a development team that is experienced and motivated to use object-oriented approaches.
Prototype	Informal development process applicable for prototypes, proof of concept, or demonstration software. Development is iterative, with minimal up-front requirements effort.
Spiral	A cyclical model of the software development process in which a repeating set of activities is performed on an increasingly more detailed representation of the product. A risk assessment must be performed at the end of each cycle and before starting a new cycle.
Waterfall	A linear model of the software development process in which the activities of each phase of the life cycle must be completed before continuing to the next phase.

With highly skilled and experienced developers on a small team, subjective evaluation criteria can be used, but for everyone else, the following advice applies: "Manage critical goals by defining **direct measures and specific targets**, assure accuracy and quality with **systematic project document inspections**, and control major risks by **limiting the size** of each testable delivery."[9]

Large projects with several teams, each having a mixture of inexperienced and skilled people, need more structure and metrics. Boehm and Turner[10] define five project dimensions that affect method and metric selection:

Size: Agile methods do not scale well. These work for projects with ten or fewer team members. Management structures, however, can incorporate teams into a large project.

Dynamism: Measured as the rate of recruitments change per month.

People Skills: The ration of highly skilled to journeyman developers.

[9] Christensen, Mark J. and Thayer, Richard H. *The Project Manager's Guide to Software Engineering's Best Practices*, IEEE Computer Society, New York, 2001.

[10] Boehm, Barry and Turner, Richard. *Balancing Agility and Discipline*: *A Guide for the Perplexed*, Addison-Wesley, Reading, MA, 2004.

Culture: The ratio of people that thrive on chaos to those people who need order to work productively.

Criticality: The potential loss of life or money because of failure.

The primary responsibility for a manager is to regularly review metrics and take action based on them. If the manager is unwilling or unable to interpret the metrics, it would be better to never require them. Forcing people to accumulate data from which there will never result analysis and action leads to low morale and problematic accuracy. Figure 1.1 shows the degree of software engineering needed under various circumstances, and it can be assumed that the degree of metrics would be comparable.

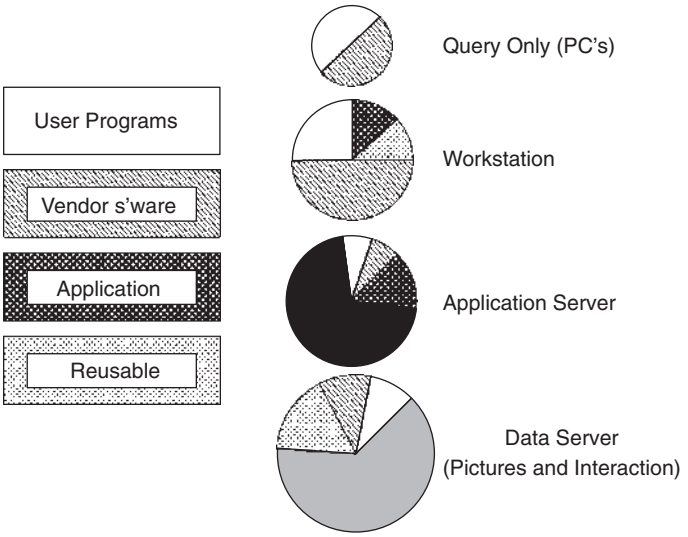

Figure 1.1. *Degree of software engineering needed under various circumstances.*

Another way of thinking about choosing a process is to look at the results that can be expected from any given process. Six processes follow, each with an explanation of the core element that caused their creation, along with a rationale of when each might be feasible.

1.5.1 No-Method "Code and Fix" Approach

Programmers sketch an idea in their heads or on a napkin and then immediately code and test at their workstations. Once they have a program that satisfies them, they give it to the customer. The customer is left to integrate the solution. There is no overhead, so the programmer can respond quickly. This approach is undisciplined, although many programmers have used it. It was

the way they learned to program in their earliest classes, or it was the way they experimented with their hobby systems. This is not to be confused with agile methods, which are disciplined. This no-method process is simply Code → Use → Debug → Fix.

This approach results in unstable systems. It can work in the short term but is not sustainable as projects grow. Although it may satisfy immediate user needs, it does not keep up with evolving user requirements. Other methods and processes are in reaction against this approach. The reason it is dignified with a name and listed first is that it is still used in crises when the chosen process breaks down. It is important to recognize this process for what it is—the bankrupt choice born of desperation.

1.5.2 Waterfall Model

In the early days of software development, computing resources were scarce, so the Waterfall Model was formalized to reduce the computer time needed to develop and test a program (Figure 1.2). The scarcity of computer time forced reliance on documents—lots of them, all subjected to meticulous review—to organize thinking and chart progress. The steps of the method are separated by the production and review of documents. Software development can proceed through these steps; other steps may be added or deleted. Extensive literature exists on the Waterfall Model that will not be repeated here.

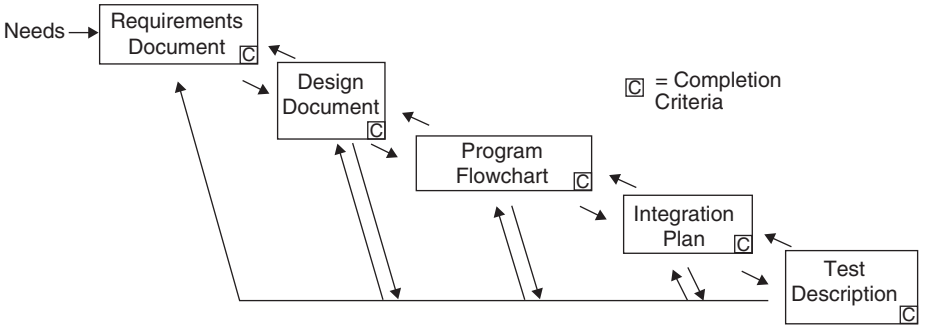

Figure 1.2. *Stages of the Waterfall Model.*

The Waterfall Model proved to be too document-intensive. The time needed to update documents often exceeded the time needed to update the code, which caused documents to be out of date and negated their value. This careful systematic approach led to long development cycles. Companies that needed short time-to-market cycles abandoned the dogmatic Waterfall Model even as they kept elements of it deep within their development methods. Last, the need for elaborate documentation tends to discourage the use of COTS, which increases development costs, slows development, and lengthens delivery time. On the other hand, high license fees of COTS components can drive up the

cost of systems. This concern encourages the use of internally developed components or open-source modules.

MAGIC NUMBER!

A good practice is to have no more than 20% of the ultimate staff participate in the requirements and architecture steps of the Waterfall Model.

A key assumption in the Waterfall Model is that requirements can be defined and carefully controlled. Unfortunately, in real life, requirements emerge as the project proceeds. That realization forced further evolution, as Barry Boehm, the prophet of the Waterfall Model, quickly realized.

Case Study: The Case of the Mandatory Requirement

Let us return to the modest store-and-forward message switching system from the *Case of the Sacrosanct Date*. We have now chosen to use the Waterfall Model. The customer's requirements (remember that a key assumption of this process is that requirements can be defined and carefully controlled) are as follows:

Requirement 1: Do not lose a message.
Requirement 2: 10-second response time is critical.

Concerning Requirement 1, the software designers were told that if they were uncertain about losing a message, they should cause the system to stop. But in a mission-critical system, there is an implied requirement that the system should not stop. High availability is needed. Availability is calculated as Mean-Time-to-Failure/(Mean-time-to-Failure + Mean-time-to-Repair).

Here the mean time to repair is the time to recover any messages that might be lost, save them, and relaunch the software. The design choice that preserved every message caused a system availability requirement to emerge.

Question: How do you resolve this contradiction?

Answer: Implicit requirements like availability are often forgotten during the writing of requirements and the implementation of a system. Sometimes to avoid a complicated design flaw, it is necessary to relax what at first seems to be a firm requirement.

Question: To satisfy Requirement 2 and reduce system costs, the architects decide to have a common buffer pool for those messages coming into the system and those going out. The input buffer requests are given higher priority to preserve the 10-second response time, which allowed users to always input messages. This solution worked and gave the required response time until more output buffers were needed than were available. The system then hung as the input process captured all the buffers. Now what can you do?

Answer: The right solution is to invert the priorities of the system. Instead of "Accept input—Drain output" change to "Drain output—Accept input." The customer was only convinced that this change was wise after seeing the new buffer request strategy demonstrated in a controlled prototype environment where that was the single change made and the system was driven with high traffic loads.

MAGIC NUMBER!

Only 40–60% of the system requirements are known at the start of the project. The rest emerge from studies of system use. Barry Boehm coined the phrase "emergent requirements" to describe them.

1.5.3 Planned Incremental Development Process

Staff and resources are never infinite. To compensate for less than ideal, i.e., normal, conditions, this process evolved to allow the separate packaging of individual functions. Here time is not necessarily critical. The idea is to divide the project into parts and apply the Waterfall Model to each part, which is sometimes called the incremental development method or the parallel development process. Each increment becomes a project unto itself, the ultimate system emerging when all the individual projects are in place.

1.5.4 Spiral Model: Planned Risk Assessment-Driven Process

The Spiral Model grew out of the incremental model, its advantage being that at each point, we have a partial view of the product. Each cycle in the Spiral could be one stage of the project processes described in the Waterfall Model. The focus is on continuous risk assessment. At each cycle, project risks are reassessed and plans are modified to contain them. The Spiral Model reduces risk, allows feedback, and is not derailed by failure because individual failures are small and constrained by the method (Figure 1.3).

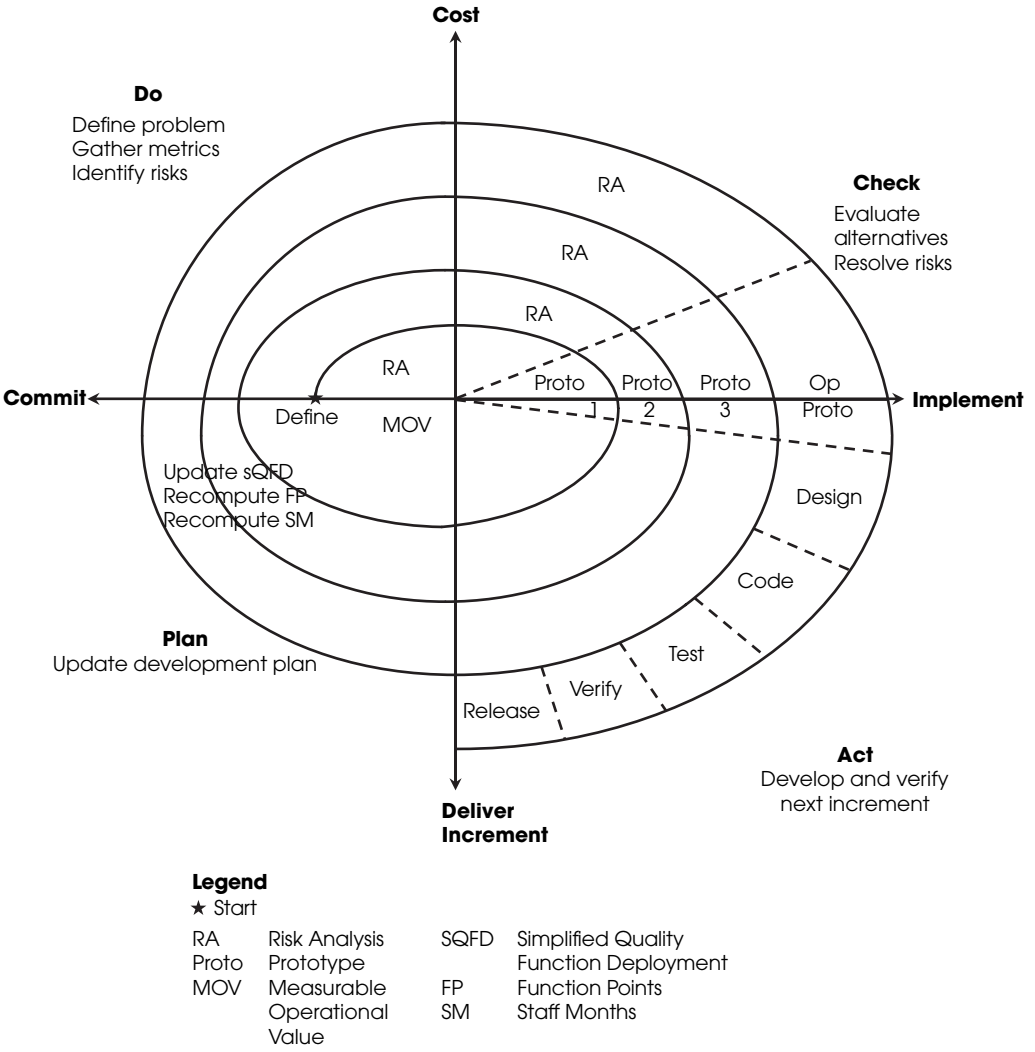

Figure 1.3. *Boehm's Spiral Model of the software process.*

Case Study: The Case of the Threatened Bottom Line

You are the project manager of a 20-person team. Your customers love your product. Suddenly, a competitor offers a product with similar features but for half your price. You do an analysis and determine that the only way to meet the price point is to reduce the team to ten people (which can be accomplished by moving ten people to another exciting project).

Question: How do you maintain quality with only half the staff?

Answer: Run your problem through several spins of Boehm's Spiral Model.

Spiral Round 1

Objective:	Identify cost reduction items
Constraints:	Meet annual profit estimates Meet customer commitments
Alternatives:	1. Object-oriented libraries → increased productivity 2. Object-oriented database → increased productivity 3. Find an existing product and tweak it 4. Improve organization/communication issues
Risks:	1. Object-oriented database is not widely used 2. Difficult to quantify improvements 3. Customers may not accept the change
Risk resolution:	1. Have a pilot or prototype 2. Do a cost analysis 3. Hire a consultant 4. Benchmarking
Result:	We discover that alternative 1 is the best
Discoveries:	It takes 6 months for programmers to learn the object-oriented approach Investment: $10,000 (software tools) + $10,000 (hardware) object-oriented databases (alternative 2) are not mature
Plan for next phase:	Introduce object-oriented technologies

Spiral Round 2

Objective:	Double to triple maintenance productivity
Constraints:	Payback or profitability has to happen within 1 year Results must be scalable to other projects Feature deployment has to be reduced from 2 years to 3 months
Alternatives:	1. Variety of compilers, debug tools, test tools 2. Workstations
Risk:	1. Investment costs 2. Finding an expert may be difficult 3. One team resists object-oriented approach
Risk Resolution:	1. Jump-start with an object-oriented expert in the team 2. Adopt older, more stable tools 3. Drop object-oriented databases 4. Develop features in parallel in C
Plan for next phase:	Build the next version of the software using object-oriented techniques Have monthly progress reviews

Spiral Round 3

Objective:	Calibrate productivity achieved Deploy first object-oriented team
Constraints:	Schedule Features and requirements
Alternatives:	1. Procedural programming 2. Object-oriented approach
Risk:	1. Compiler may produce unreliable code 2. Six-month learning curve
Risk resolution:	1. Develop critical features in C 2. Train team 3. Add object-oriented experts 4. Reduce reporting
Plan for next phase:	Integrate into development plan
Results:	You attracted better staff. Customer got desired quality and price. Less documentation, more comments. Code reuse doubled. 30% less code to write.

Interestingly, even though change costs are a third of what they were, there was only a 50% overall cost reduction because code changes accounted for 70% of the costs: $0.3 + (0.7 \times 0.33) = 0.53$.

The Spiral Model demands knowledge of project objectives, constraints, and preferred architectures. The customer knows only the features required to do a subset of the job—the subset the customer thinks is most important. Special aspects of the problem, well known to problem domain experts, are frequently unstated. Ideally, the developer would simply ask the customer what was required and the customer would provide sufficient detail to proceed; the customer would be totally familiar with the expert's unstated constraints. This expectation is unreasonable, and significant negotiations between both parties are required to balance functionality, performance, and reliability with cost and schedule considerations.

Boehm's WinWin Spiral Model, shown in Figure 1.4, derives its name from the objective of these negotiations. The client wins the product that satisfies most needs, and the developer wins by working to realistic and achievable budgets and deadlines. To achieve this objective, the model defines a set of negotiation activities at the beginning of each pass around the spiral. The following activities define the customer communication:

Identification of the system stakeholders—those in the organization that have a direct business interest in the product to be built and will be

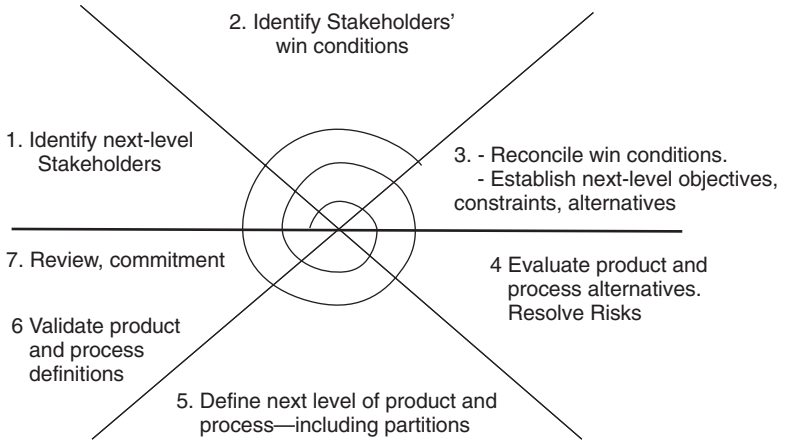

Figure 1.4. *Boehm's WinWin Spiral Model.*

rewarded for a successful outcome or criticized if the effort fails (e.g., user, customer, developer, maintainer, interfacer, etc.).

Determinations of the stakeholders' "win conditions."

Negotiations of the stakeholders' win conditions to reconcile them into a set of win–win conditions for all concerned (including the software project team).

In addition to the early emphasis placed on the win–win condition, the model also introduces three points that help establish the completion of one cycle around the spiral and provide the decision milestones before the software project proceeds:

Life-Cycle Objectives (LCO): Defines a set of objectives for each major software activity (e.g., a set of objectives associated with the definition of top-level product requirements).

Life-Cycle Architecture (LCA): Establishes the objectives that must be met as the software architecture is defined.

Initial Operational Capability (IOC): Represents a set of objectives associated with the preparation of the software for installation/distribution, site preparations before installations, and assistance required by all parties that will use or support the software.

The WinWin process is a framework for distributed asynchronous decision making when there are many stakeholders. It provides a model useful for negotiation. It allows architecture constraints to override a particular requirement once the stakeholder understands its cost or schedule impact. This approach is similar to the agile method of intimate customer interaction in the same workplace. WinWin provides a structured model for capturing all

stakeholder concerns. WinWin speeds software development by eliminating rework that occurs from misunderstandings and results in better products by exposing architecture options early in the development.

1.5.5 Development Plan Approach

An effective process for large projects requiring more than 10 people is to write a development plan that answers these questions and is updated at the end of each step:

What will you do?
How will you do it?
What do you depend on?
When will you be done?
Who will do what?

All disciplined methods recognize the need for these steps, if not in this order or with the same mandatory documents. Therefore, all tasks needed to deliver the software product should be identifiable. It is most useful to transform your task list into a table similar to the one shown in Table 1.2 that will reflect the project schedule.

TABLE 1.2. Development Plan Chart

Task	Who	When	Current Estimate

Task: A description of what is to be done with a measurable output
Who: The person in charge of the task
When: Planned completion date that can be modified only by the project manager
Current Estimate: Estimated completion date by the "Who" person. This unique feature makes this more useful than any PERT or Gantt chart.[11]

The use of the "current estimate" date permits task owners to report trouble in a way the rest of the project can understand without upsetting the entire project plan. Without the current estimate feature, a late task could cause total project replanning when the manager might have been able, with early warning, to resolve the problem without delaying project completion. The early warning of the current estimate feature allows developers to be honest about where they stand while giving the project manager time to expedite or replan incrementally.

The core element of the development plan is hierarchical control via the itemization of concrete signposts that indicate to every project member the health and condition of the project (Figure 1.5). This nonjudgmental

[11] L. Bernstein developed this approach while working for Victor Vyssotsky at Bell Laboratories. Mr. Vyssotsky described it to Fred Brooks, who was pleased to use it in his book, *The Mythical Man-Month*.

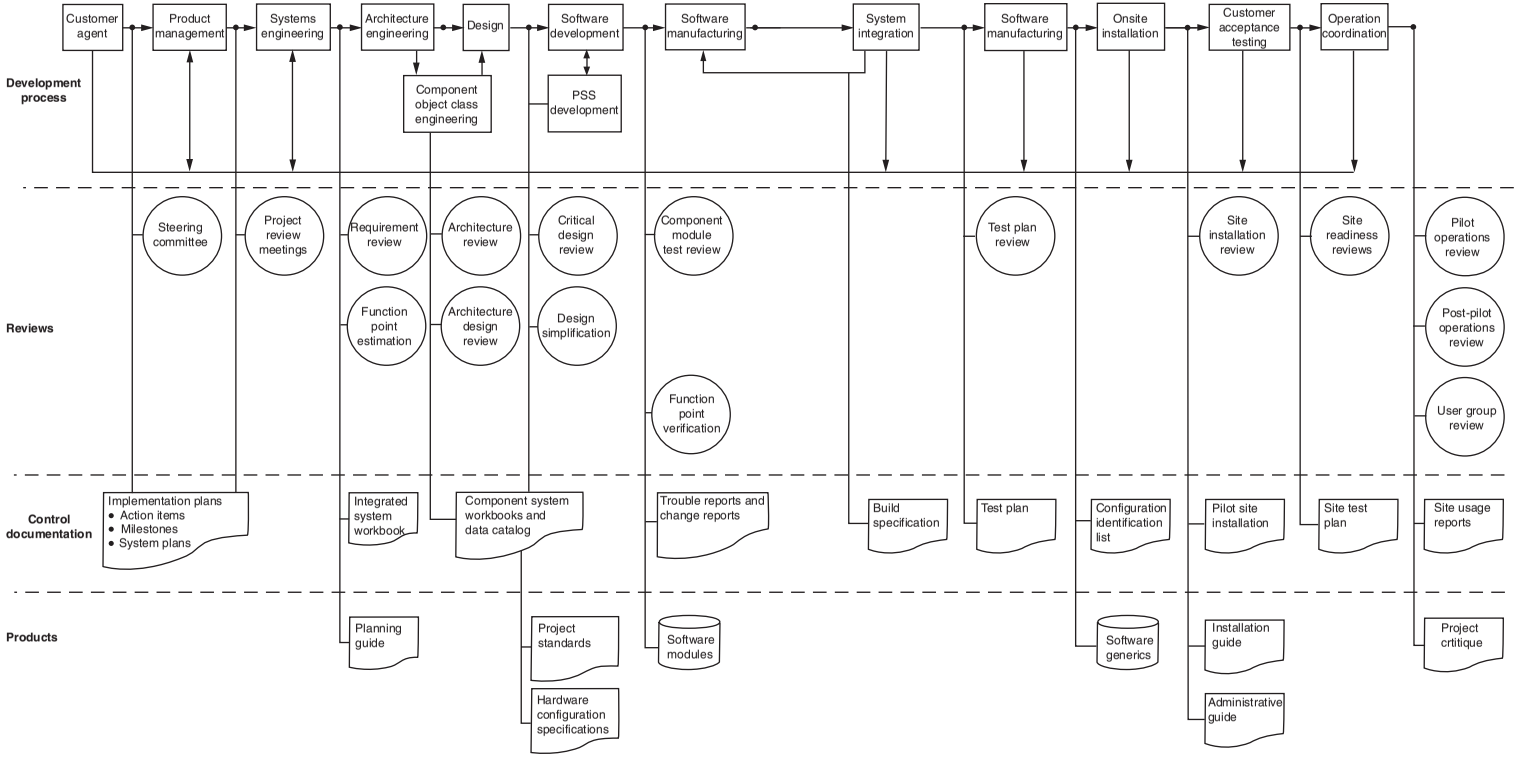

Figure 1.5. *Development plan approach.*

reporting tends to focus all team members on the person's responsibility to the larger objective of project completion. The hazards of this method are that it may lead to functional boundaries that are too sharply defined, that task divisions may lead to suboptimization, or that it may discourage risk management. Figure 1.5 is an elaborate schematic, used to manage a project with 500 people. Select only those elements that fit your project.

1.5.6 Agile Process: an Apparent Oxymoron

Agile advocates are not antimethodology. They embrace modeling, but not merely to file some diagram in a dusty corporate repository. They embrace documentation, but not to waste reams of paper in never-maintained and rarely used tomes. They insist on planning, but recognize the limits of planning in a turbulent environment. The core element that agile process proponents emphasize is the schedule. It is the most important criterion of success. Agile methods drive organizations to schedule containment.

Those who brand proponents of XP, SCRUM, or any of the other agile methodologies as hackers are ignorant of both the methodologies and the original definition of the term (a "hacker" was first defined as a programmer who enjoys solving complex programming problems, rather than someone who practices ad hoc development or destruction).[12]

Early proponents of agile methods felt sufficiently beleaguered by the rigid enforcement of oppressive process that they actually voiced a "manifesto" declaring their values. These were as follows:

Individuals and interactions over processes and tools
Working software over comprehensive documentation
Customer collaboration over contract negotiation
Responding to change over following a plan

If we look at one agile process, extreme programming (XP), we find that it is highly disciplined. It is not chaotic, even though some slipshod programmers hide behind the term and slovenly software shops describe their approach as XP. On the contrary, it involves considerable emphasis on disciplined planning:

Close and teaming customer interactions
Documented user stories and use cases
Paired programmers working closely together
Testing before coding
Simplicity in design
Relatively frequent small releases
Iteration planning

[12] www.agilemanifesto.org/hisoty.html.

XP demands devotion to agreed-on standards, frequent integration, deferred optimization, and unit tests before integration release and for every bug. Acceptance tests are run often, and the results are published and made available to all project team members, including the customer.

1.6 REEMERGENCE OF MODEL-BASED SOFTWARE DEVELOPMENT

This approach is from the early days of large system development when computer resources were limited and expensive. With the need to use components to speed software development, the use of architecture patterns, frameworks, and models became popular. Components could be evaluated in terms of how well they fit into an integrated software system.

The Model-Driven Software Development (MDSD) paradigm supports development teams with 20 or more developers. It evolved from software product line engineering where families of applications for a specific market segment are built. Emphasis on agile software development sets MDSD apart from the earlier Waterfall Model, planned incremental development, and the Spiral Model. MDSD, like prototyping, produces working software early in the development cycle, but MDSD also provides the scalability that is not inherent in popular agile methodologies.

The Safeguard Anti-missile Missile System in 1968 is an example. Developing its software was a major challenge. No configuration control guidelines or standard software processes existed. The National Academy of Sciences opined that such software could not work. Only NASA faced similar complexity and performance demands. Although Safeguard borrowed many techniques from NASA's Apollo project, a unique approach to software development evolved spontaneously. Many projects modeled and simulated system performance, but Safeguard was the first to use the models and simulations to drive its software development. After some early false starts, the entire software effort fell into lockstep with the model and simulation program.

Once the decision was made to develop a tactical anti ballistic missile (ABM) system, a system evaluation department was formed to provide quality assurance. Its objective was to ensure that the design met the system objectives and that the implementation met the system requirements.[13] To carry out that objective, the department designed field tests. This effort emerged as so important that it became the driving force for the software development effort. Typical mission scenarios measured software progress throughout the entire development cycle. These scenaries are now called use cases.

[13] Bernstein, L., Burke, E. H., and Bauer W. F. "Simulation- and Modeling-Driven Software Development," *Crosstalk: The Journal of Defense Software Engineering*, Vol. 9, No. 7, July 1996, pp. 25–27.

The system evaluation team took the approach of developing a family of simulations to predict and confirm system performance. The highest-level simulation predicted the performance of the entire system to a full-scale attack. To facilitate the design of the simulation, the models of subsystems (missiles and radars) were only as detailed as was required to enable the system simulation to model overall system performance. Detailed simulations of all major subsystems validated the high-level models. In some cases, the phenomena modeled in those subsystem simulations were based on even more detailed simulations accounting for the fundamental physics involved.

Being able to produce several models simultaneously for different subsystems is fundamental to model-based development. The appropriate model iterates and validates each small increment of the system. There is no attempt to create a single, all encompassing model for the entire scope of the desired system. Models are merely abstract representations of software and therefore may not be completely accurate. They must be validated with realistic data, but the focus should remain on only the uncertain aspects of the subsystem. Attempts to create a highly detailed model should be avoided; good engineering judgment and problem domain expertise are essential for deciding just how detailed the model needs to be. Test cases evolve from the modeling efforts. The major benefit of model-based development is that discrepancies between the model and the test results efficiently point to specific areas to correct.

During the 1980s and 1990s, projects became schedule-driven to the exclusion of other needs. Faster time-to-market became the way to success, and models were seen as delaying product deployment. As the need for trustworthy systems captured the imagination of the public, model-based development gave developers a way to understand safety, reliability, schedule, and performance concerns while meeting their schedule commitments.

1.7 PROCESS EVOLUTION

Suppose one or another of the processes described appeals to a design team, or a new technology bursts on the scene, but middle management seems to stonewall any suggestions to adopt new ways. Software developers and company executives are favorably disposed toward anything that speeds development and reduces bugs, but there is no incentive for middle managers to risk using a new approach. This position is astonishing initially, especially if those middle managers were themselves software developers in recent memory, but, on examination, it is perfectly rational. Training and staffing are cost items in their budget against a method with which they have no experience. If their organization does not deliver, they are to blame. If they are successful, the executives take the credit for having the vision to move to a new technology. So how can an organization ever evolve to using more sophisticated processes?

The answer is to invest in corporate-funded investigations of technology, training, and shared risk. Companies can move to new technologies by making

informed decisions. An organization with more than 250 technical people needs a well-defined "someone" to encourage the adoption of new software processes.

An office of technology planning (OOTP) could encourage the adoption of new technology. Table 1.3 shows how such an office might work. The Chief Technology Officer (CTO) leads the office and has some discretionary funds for evaluating, trialing, and deploying technology throughout the organization, which plugs the drain on middle managers' profits as their people's skills are improved. The CEO determines how much the company can and should invest and the CTO makes the investment pay off.

TABLE 1.3. Technology Deployment Processes

Market Input Ideas Technology Assessment	→	**Technology Initiatives** Metric: percent of new product revenue for the last three years and royalties	→	Processes & Products with Embedded New Technology

Technology Initiatives Subprocesses

Technology Plan Development	Acquire Technology (Make/Buy)	Prioritize Applied Research	Contract Applied Research Initiatives	Deploy Technology

FUNCTIONS

CEO	— Review technology plan for consistency with strategic plan
CHIEF TECHNOLOGY OFFICER	— Provide technology vision — Facilitate and accelerate technology transfer/diffusion — Lead the Office of Technology Planning — Recommend technology initiatives — Integrate 2- to 5-year product line initiatives — Ensure consistency with personnel policies — Chair Intellectual Property Study team
OFFICE OF TECHNOLOGY PLANNING	— Drive to asset-based business — Professional societies — Recommend technology priorities — Manage and implement the technology plan — Assess R&D capability and technology for benchmarking
Tool Providers	— Provide technology base and roadmap technologies

Bernstein was the CTO from 1991 to 1994 for the Bell Laboratories network management development organization comprising 2000 people. The technology transfer process was aimed at creating new products and services for the customer base and for making software development more effective and cheaper. Input came from customers, sales teams, internal and external researchers, developers, and formal, chartered technology assessment efforts.

The OOTP held monthly meetings that the CTO chaired to determine technology priorities. For example, a study of object-oriented database technology

in 1992 showed it to be too immature for wide use. Two projects stopped using it before they reached the point of firm commitment to the method. On the other hand, tools to find and fix memory leaks were found to be effective and were deployed to every project within 4 months of completion of the technology assessment report. The OOTP managed the technology budget for the organization, funded initiatives, and produced a technology plan. It also managed the adoption of the technology project by project, providing experts as needed to expedite adoption of the new tool, process, or component. The ratio of revenue from new products doubled in the interval, while productivity increased fourfold.

1.8 ORGANIZATION STRUCTURE

There are two possibilities for deploying the staff talent that any company engaged in software products must confront. Each has its benefits and drawbacks. The functional organization is efficient because each functional group does only its job for every product produced. For example, people expert at writing requirements will do only that for every product produced. The drawback to this style is that communication problems can develop among functional groups. Also, a sense of the whole product, and the concomitant sense of satisfaction in its ultimate success, is lacking. Project organization is less efficient but generally more effective in encouraging personal growth and reducing professional burnout. This style does foster a commitment to the success of the project as a whole and a sense of broad responsibility to the customer.

No right answer exists. Functional organization is efficient for manufacturing an established product. Project organization tends to be successful for software, especially when either the problem or the technology is not well understood. Firm commitment to one or the other style of organization is best made after a working prototype is achieved and risks are assessed (Table 1.4).

TABLE 1.4. Functional vs. Project Organization

Functional	Project
Requirements	Customer interface
Design	Program design
Code	Integration
Test	Validation and verification

Project organizations form teams more easily because their members are focused on a common goal. In functional organizations, most people are assigned to several teams to take advantage of their special skills. As a result, they can be distracted when there are unexpected problems and some of their teams are necessarily shortchanged.

What is the difference between a mediocre and a championship team? Is it train-train-train, secure the best talent available, or devise innovative strategy?

These are important, but the most important factor is commitment to quality. Members of a championship team, whatever the game, share similar qualities:

Want to be in the game. They are not content to sit on the sidelines and watch the action; they want to participate in the effort and share in the victories.

Are highly and visibly enthusiastic. It is contagious. Contributions are recognized and rewarded, resulting in increased confidence.

Desire to be top performers and realize that open and honest feedback among coaches and other team members is critical to individual and group success. Asking, "How am I doing?" or "How can I do my job better?" ensures that opportunities are available to always improve knowledge and skills.

Use the word "we" rather than "I," realizing that strategies are developed for group action.

Respect the talents and abilities of fellow teammates, while analyzing the strengths and weaknesses of rivals.

Understand the value of communication—that it is important to know where the team is going and how individual action can attain the goal.

Anticipate change and react quickly while continuing to drive toward the goal.

Have confidence in their team, their managers, and themselves.

Another dimension of organization structure is hierarchical versus network. Hierarchical organizations reduce the number of communication paths among people. Pair-wise communication requires n(n-1)/2 between n people. A small shop of 7 people requires 21 paths, so everybody tends to know what is going on. A shop of 50 people would require 1225 paths, a much more difficult situation for maintaining a fully informed team.

A hierarchical organization might group together everyone defining requirements in one group, those creating the architecture in another, software developers, human/computer interface experts, testers, and integrators in yet other discrete groups. Each group would have a supervisor reporting to next-level managers, and so on to the highest corporate executive. If everyone was focused on a single product, it would be both a project and a functional organization of the hierarchy. More likely, specialized skills would be shared by more than one product line. To understand how an organization really works, a network view is necessary. Software manufacturers are the common communication point that pulls together each product line and reduces the number of communication paths. Figure 1.6 illustrates this.

Software manufacturing is a systematic approach to system building, deliverable documentation production, configuration identification, change control, and packaging for delivery. Software manufacturing is in-line, not overhead. Software developers do not do the job of system building; people with pro-

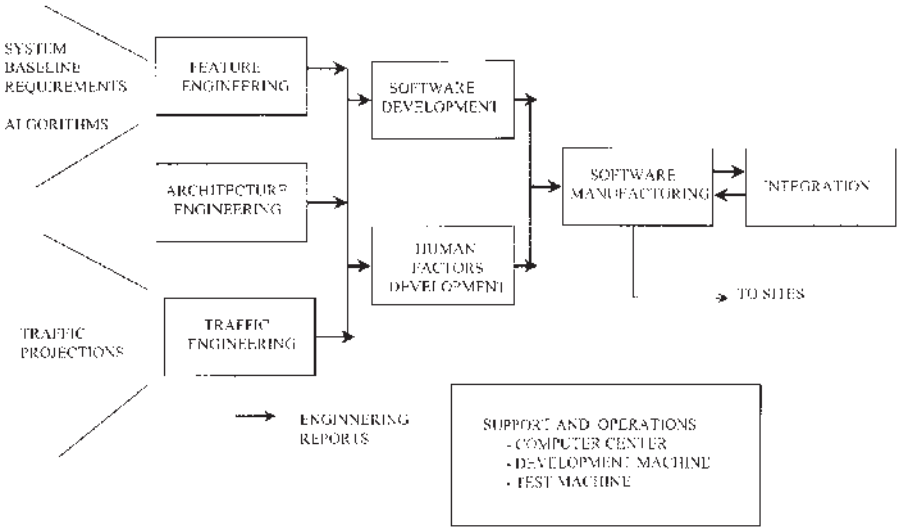

Figure 1.6. *Network organization.*

duction skills and specially trained expediters, in numbers amounting to approximately 5% of the programming staff, build systems.

The software manufacturing group runs the computers for the project. All releases to system integration and the field must go through software manufacturing. When schedules get tight and the resident wunderkind is sure that running to the customer with a fix or two will save the day, the discipline of software manufacturing stands between fiscal sanity and chaos.

1.9 PRINCIPLES OF SOUND ORGANIZATIONS

People make the difference. Hire good people and respect their individual talents. People are not interchangeable. Software people thrive on challenge and new technology. Some experts can produce ten times more than journeyman programmers. Because customer service and innovations are increasingly important competitive weapons, it does not pay to create a sullen, dispirited, or burned out workforce.

Here are some ways to "turn down the heat:"

Work is NOT an endurance contest.

Be flexible in work hours.

Ask your people how to restructure the work.

Reserve overtime hours for true crises, not part of the standard day.[14]

[14] O'Reilly, Brian. *Fortune Magazine*, March 12, 1990.

Make the customer and your supplier your teammates and you can halve the development time by reducing confusion and misunderstanding. Before the first release, customers want lower prices, more features, and shorter schedules. When the system becomes operational, they want better reliability, throughput, and response time. Your suppliers can ruin your hard-won teaming relationship. To make sure that your suppliers understand the importance of your customer, it is vital that you let them participate in all phases of the project.

Understand the requirements: Validate, prototype, and make sure there are numeric operating and bounds specifications for every requirement. Customers **need** some features and **want** others. A successful project manager distills the needs from the wants and satisfies the needs while delivering a reliable system at reasonable cost. Firm commitments are best made after a prototype works.

Configuration management, the control of changes to the software, is essential in all projects, large and small. Quality and simplification in the earliest steps save time and money.

Finally, none of these will work unless our profession recognizes the next core element in the evolution of software processes as a fundamental principle. Software trustworthiness is the next major area in which academia and industry must focus—both for national security reasons as well as to ensure that the U.S. software industry maintains its leadership. The three attributes of software reliability, security, and safety comprise trustworthiness. Fostering these attributes can spark another round of American software innovation that will stem the wholesale outsourcing of our industry.

Companies and people will act to assure trustworthiness only if there are economic consequences to not doing so. Evaluation and liability, both corporate and individual, are the only means to bring awareness to the impact of negligence. Software trustworthiness is critical for medical devices, plant control systems, and weapon systems; for management of sensitive records; and for critical infrastructure. Dependability is fundamental to cybersecurity.

Doctors and lawyers are members of old professions and as such recognize that being an important part of many peoples' lives carries with it ethical responsibility and liability. This course will help make you ready to assume your responsibility for trustworthiness in our young profession.

Case Study: The Case of the Dissed Discount

Once upon a time, a computer manufacturer offered 10% price discounts for a volume purchase. The cost of the computers was 25% of the cost of the system.

Question: The value of the system to the customer was 50 times the cost of the system if it could be available by a critical date. Would you accept the discounts and run the risk of having no leverage over your supplier's scheduling? If not, what else might you do?

Answer: In this case, the best strategy was to let the manufacturer charge list price, but to insist on premium service, delivery without a contract in hand, and a guarantee to have spare parts on-site.

The manufacturer shipped the computer to the operational site while the customer's purchasing organization processed the paperwork. Insurance concerns prevented them from installing the computer, but they could leave it on their moving van, and they did. They left the van containing the computer sitting at the loading dock for 2 days, and the installers were kept on immediate call. Within 15 minutes of getting the signed contract, the waiting installers set to work and had the computer operational in record time. This saved the normally scheduled 2-week installation time and helped make up for other schedule slips.

The moral of the story is that you must be a world-class customer before you can be a world-class supplier!

1.10 SHORT PROJECTS—4 TO 6 WEEKS

1.10.1 Project 1: Automating Library Overdue Book Notices

An elementary school of 500 students is using a manual method for tracking books on loan. As books are taken from the library, the book card is taken from the book jacket and filed by date. Books may be borrowed for 2 weeks. As books are returned, the card is put back into the book jacket. Books that are not returned in 2 weeks are considered overdue, and an Overdue Book Notice is sent to the students. Students who do not return a book within 3 weeks are given a second notice. Books not returned in 4 weeks have a third notice and are reported on a special "Critical Overdue" librarian's report. This report is sent to the student's teacher and to the principal. The book card is given to the librarian.

Clerks write overdue book notices on half-sheets of paper for about 200 books each week. These are distributed weekly. This form has been used for many years:

Glenwood School Overdue Book Notice
Book Title:
Student Name:
Teacher Name:
Date of Notice:
Notice 1 2 3 (circle one)
This book is overdue. Please return it promptly.

PROBLEM:

Automate the overdue book notice process with a computer borrowed from the computer class. The computer room is next to the library. This is the first

computer automation project in the school. Data may not be left on the computer from week to week. The computer is not networked.

1.10.2 Project 2: Ajax Transporters, Inc. Maintenance Project

Ajax Transporters, Inc. manufactures exactly one size of one model of one product, the Ajax Personal Transporter, which they sell directly to customers. Ajax's computer system consists of an order entry (sub-)system and an inventory/order fulfillment (sub-)system ("inventory system," for short).

Orders arrive by phone or by mail and are entered into the order entry system by clerks. The order entry system's main, although not its only function is to check the validity of each incoming order, retain a copy of each valid order in its database, and send valid orders to the inventory system.

The inventory system checks whether there are enough personal transporters to satisfy an incoming order. If there are, the warehouse workers remove the required number of personal transporters from inventory and bring them over to Schlepper Shipping, Inc., which ships the order to the customer. The inventory system also sends a message to the order entry system indicating that the order has been filled. If there are not enough personal transporters to fulfill the order, the inventory system puts the order into its database as a "back order." Warehouse workers check back orders each day and fill as many as possible, given inventory on hand, which is increased whenever Ajax's Manufacturing Division manufactures more personal transporters and delivers them to the warehouse.

Ajax currently uses a COTS software package for its order entry/inventory system. It is manufactured by So-So Software, Inc. Because it is a proprietary product, it is distributed only as an executable; nobody outside of So-So gets to see the source code.

The following transactions are recognized, from terminals or the website, by the order entry/inventory system:

Place New Order

View All Orders

Cancel Order by Order Number

You have access to the system's main customer/order file. Many pages detail the transactions and access to the file.

PROBLEM:

The system is working well, and Ajax's sales are increasing when they are told that Schlepper Shipping, Inc. is going out of business because of the economic downturn. The U.S. Post Office agrees to contract for the shipping, but So-So Software, Inc. has accidentally forgotten to include the customer name with other customer information in the file and the Post Office will do the shipping only if the customer name is added to the address. Ajax has asked So-So Soft-

ware to make this change in the system, but So-So wants far more money than Ajax feels is reasonable to make so small a change.

You work for Amber Consulting, Inc., and your group at Amber has just been given the job of making the change to Ajax's system, but without changing the COTS product currently in use. (You will do this work by building one or more new modules that interact with the COTS system.) Ajax is getting one of their in-house groups to completely rewrite their system, but it needs the upgrade that you have been contracted to do well before the rewritten system can be delivered because they need to continue doing business or they will go bankrupt. Your task is to upgrade the system in 5 weeks.

1.11 PROBLEMS

1.11.1 You have two releases of the system in the hands of four customers. Each customer wants their changes but does not want to be burdened with the changes of the other customers. Your budget is tight.

a. You insist that there is one release for all and that the customers must upgrade and accept all changes.
b. You adopt a versioning configuration management system.
c. You break the system into four systems and customize each.
d. You refer the problem to the product manager for a business analysis of the best strategy.

1.11.2 You are eager to improve the long-term productivity of the group of developers you are managing, so you

a. ask them to work overtime.
b. measure their work.
c. manage them more closely.
d. provide them with new tools.

1.11.3 Your program has worked for several months and all users are pleased with it. Suddenly it crashes, so you

a. blame the user for not being properly trained.
b. seek program dumps and begin debugging.
c. determine what changed in the run that crashed.
d. quit and find a new job.

1.11.4 You find it helpful to discuss your program with a colleague; you review each other's code. This process is called

a. code inspections.
b. code reviews.
c. extreme programming.
d. a waste of time impacting productivity.

1.11.5 You are asked to produce a program on a tight schedule. Your boss tells you what is needed. To meet the schedule you

a. begin coding immediately.
b. research other solutions.
c. discuss the program with the software architects.
d. design the interfaces.

1.11.6 You see that you need a new tool, so you

a. ask your boss to get one.
b. build it yourself because it would take too long to purchase one.
c. borrow one from a colleague by copying a CD.
d. work around getting it and stick to your development schedule.

1.11.7 When developers use the incremental model to organize their releases in into products, they

a. deliver a full set of modules with limited functionality, and then gradually change the functions of the modules with each new release.
b. deliver a subset of the modules that perform a limited set of functions, and then deliver more modules with each new release.
c. deliver the full set of functions and modules with the first release.
d. deliver a limited set of functions and wait for customer input to add functions.

1.11.8 This problem continues the first example in this chapter. A software system is designed so that after every hour of normal operation, it stops and relaunches from its initial state. This process is called software rejuvenation. Assume that the mean time to failure for the software is measured to be 10 hours and the mean time to repair is 5 minutes. Repair means restoring the software to an operating condition by relaunching it from its initial state.

a. What is the reliability of the system after 4 successful hours of execution and four successful rejuvenations?
b. What is the probability of failure of the system for any 4 hours of operation?
c. What is the availability of the system where $R(t) = e^{-\lambda t}$, where R is reliability, c is the reciprocal of the mean time to failure, and t is the execution time. Note that e^x is approximately $(1 + x)$ for small x.

1.11.9 You are asked to be the architect for the development of a customer resource management transaction system. The system must be fast so that there are no bottlenecks when the system is executing. The agents using the system are trading millions of dollars an hour so transaction speed and system availability are paramount. System cost is of secondary importance. The customer agrees to a margin of three times the required transaction speed. Once

you have a first version of the system in your integration laboratory, you run expected use case scenarios and trace transaction execution time. You find that average transaction execution uses 1 million processor instructions with a 100-instruction standard deviation. You have a balanced design in your system. The computer chosen for the production system executes an average instruction in 500 ns.

a. Assess the risks in the system architecture for the first version.
b. What design changes are needed to conform to the system performance constraints?
c. What alternatives to software redesign are possible?

1.11.10 You are the project manager for a client/server system. Your responsibility is for the server. The server software has successfully completed system test and is certified to support 100 clients. The software architect has designed a safety margin of two into the client support requirement. You know that the system will reliably support 200 clients with satisfactory performance, but because of the safety margin, the server software is rated at 100 clients. Your customer calls in a panic. A merger has occurred, and his job is on the line. His server must support 175 clients, or the company will convert to a similar server acquired in the merger. What will you do?

1.11.11 You own a software company that has several projects underway. You need to deliver a system so that you can bill your customers and meet your payroll. Your project manager for one system reports an outstanding problem list of 200 unrelated and uniformly distributed critical problems. Each critical problem is estimated to take 1 week for a tester and a developer working together to resolve. Software developers can fix problems in 2 days if they can resubmit their modules instead of submitting problem fixes. There are five modules that make up the system, and the five system testers working as a test team can fully test a module in 1 day when no critical problems are found. There are five testers and 20 developers on this project. After all problems are fixed, a 2-week regression test period is needed before product ship. Customers are willing to accept software with noncritical problems outstanding. Because the other systems are just entering system test, you need to obtain a loan to cover the time needed to get this system in shape to cover your payroll. What is the shortest time before you can ship the software and therefore the time you need to have the loan? What risks are you taking? Assume no other critical problems are found once these 200 are fixed.

BIBLIOGRAPHY

Bernstein, Larry. "Software in the Large," *AT&T Technical Journal*, Vol. 75, No. 1, Jan./Feb. 1996, pp. 5–14.

Binder, Robert V. *Testing Object-Oriented Systems—Models, Patterns, Tools*, Addison-Wesley, Reading, MA, March 2003.

Boehm, Barry W., et al. *Software Cost Estimation with COCOMO II*, Prentice Hall, Englewood Cliffs, NJ, 2000.

Crowley, Thomas. "Safeguard Data—Processing Subsystem," *The Bell System Technical Journal*, Special Supplement, 1975.

Carnegie Mellon University Software Engineering Institute. *The Capability Maturity Model: Guidelines for Improving the Software Process*, Addison-Wesley, Reading, MA, 1995.

Fowler, Martin. *Refactoring Improving the Design of Existing Code*, Addison-Wesley, Reading, MA, 2000.

Hatton, Les. *Safer C-Developing Software for High-Integrity and Safety-Critical Systems*, McGraw-Hill, New York, 1997.

Hoffman, Daniel M. and Weiss, David M. *Software Fundamentals Collected Papers by David L. Parnas*, Addison-Wesley, Reading, MA, 2002.

Jones, T. Capers. *Estimating Software Costs*, McGraw-Hill, New York, 1998.

Leavitt, Harold J. and Bahrami, Homa. *Managerial Psychology: Managing Behavior in Organizations*, 5th ed. University of Chicago Press, Chicago, IL, 1988.

Martin, Robert C. *Agile Software Development*, Prentice Hall, Englewood Cliffs, NJ, 2003.

Maxwell, Katrina D. *Applied Statistics for Software Mangers*, Prentice Hall, Englewood Cliffs, NJ, 2002.

McConnell, Steve. *Software Project-Survival Guide*, Microsoft Press, Redmond, WA, 1998.

Peters, James F. and Pedrycz, Witold. *Software Engineering—An Engineering Approach*, John Wiley and Sons, New York, 2000.

Pressman, Roger S. *Software Engineering: A Practioner's Approach*, McGraw-Hill, New York, 2002.

Van Vliet, Hans. *Software Engineering—Principles and Practices*, 2nd ed. John Wiley and Sons, New York, 2000.

Voas, Jeffrey and McGraw, Gary. *Software Fault Injection: Inoculating Programs Against Errors*, John Wiley and Sons, New York, 1998.

Wallnau, Kurt, et al., *Building Systems from Commercial Components*, Addison-Wesley, Reading, MA, 2001.