CHAPTER 1

Software Development Methodology Today

Cease dependence on inspection to achieve quality.
—W. Edwards Deming

Quality is a many-splendored thing, and every improvement of its attributes is at once an advance and an advantage.
—C. V. Ramamoorthy

Overview

Both personal productivity and enterprise server software are routinely shipped to their users with defects, called *bugs* from the early days of computing. This error rate and its consequent failures in operation would not be tolerated for any manufactured or "hardware" product sold today. But software is not a manufactured product in the same sense as a mechanical device or household appliance, even a desktop computer. Since programming began as an intellectual and economic activity with the ENIAC in 1946, a great deal of attention has been given to making software programs as reliable as the computer hardware they run on. Unlike most manufactured goods, software undergoes continual redesign and upgrading in practice because the system component adapts the general-purpose computer to its varied and often-changing, special-purpose applications. As needs change, so must the software programs that were designed to meet them. A large body of technology has developed over the past 50 years to make software more reliable and hence trustworthy. This introductory chapter reviews the leading models for software development and proposes a robust software development model based on the best practices of the past, while incorporating the promise of more recent programming technology. The *Robust Software Development Model (RSDM)* recognizes that although software is designed and "engineered," it is not manufactured in the usual sense of that word. Furthermore, it recognizes an even stronger need in software development to address quality problems upstream, because that is where almost all software defects are introduced. *Design for Trustworthy Software (DFTS)* addresses the challenges of producing trustworthy software using a combination of the iterative *Robust Software Development Model*, *Software Design Optimization Engineering*, and *Object-Oriented Design Technology*.

Chapter Outline

Software Development: The Need for a New Paradigm

Computing has been the fastest-growing technology in human history. The performance of computing hardware has increased by more than a factor of 10^{10} (10,000 million times) since the commercial exploitation of the electronic technology developed for the ENIAC 50 years ago, first by Eckert and Mauchly Corp., later by IBM, and eventually by many others. In the same amount of time, programming performance, a highly labor-intensive activity, has increased by about 500 times. A productivity increase of this magnitude for a labor-intensive activity in only 50 years is truly amazing, but unfortunately it is dwarfed by productivity gains in hardware. It's further marred by low customer satisfaction resulting from high cost, low reliability, and unacceptable development delays. In addition, the incredible increase in available computer hardware cycles has forced a demand for more and better software. Much of the increase in programming productivity has, as you might expect, been due to increased automation in computer software production. Increased internal use of this enormous hardware largesse to offset shortcomings in software and "manware" have accounted for most of the gain. Programmers are not 500 times more productive today because they can program faster or better, but because they have more sophisticated tools such as compilers, operating systems, program development environments, and integrated development environments. They also employ more sophisticated organizational concepts in the cooperative development of programs and employ more sophisticated programming language constructs such as Object-Oriented Programming (OOP), class libraries, and object frameworks. The first automation tools developed in the 1950s by people such as Betty Holburton[1] at the Harvard Computation Laboratory (the sort-merge generator) and Mandalay Grems[2] at the Boeing Airplane Company (interpretive programming systems) have emerged again. Now they take the form of automatic program generation, round-tripping, and of course the ubiquitous Java Virtual Machine, itself an interpretive programming system.

Over the years, a number of rules of thumb or best practices have developed among enterprise software developers, both in-house and commercial or third-party vendors. Enterprise software is the set of programs that a firm, small or large, uses to run its business. It is usually conceded that it costs ten times as much to prepare (or "bulletproof") an enterprise application for the marketplace as it costs to get it running in the "lab." It costs another factor of 2 from that point to market a software package to the break-even point. The high cost of software development in both time and dollars, not to mention political or career costs (software development is often referred to as an "electropolitical" problem, and a high-risk project as a "death march"), has encouraged the rise of the third-party application software industry and its many vendors. Our experience with leading both in-house and third-party vendor enterprise software development indicates that the cost of maintaining a software system over its typical five-year life cycle is equal to its original development cost.

Each of the steps in the software life cycle, as shown in Figure 1.1, is supported by numerous methods and approaches, all well-documented by textbooks and taught in university and industrial courses. The steps are also supported by numerous consulting firms, each having a custom or proprietary methodology, and by practitioners well-trained in it. In spite of all of this experience supported by both computing and organizational technology, the question remains: "Why does software have bugs?" In the past two decades it has been popular to employ an analogy between hardware design and manufacture and software design and development. Software "engineering" has become a topic of intense interest in an effort to learn from the proven practices of hardware engineering—that is, how we might design and build bug-free software. After all, no reputable hardware manufacturer would ship products known to have flaws, yet software developers do this routinely. Why?

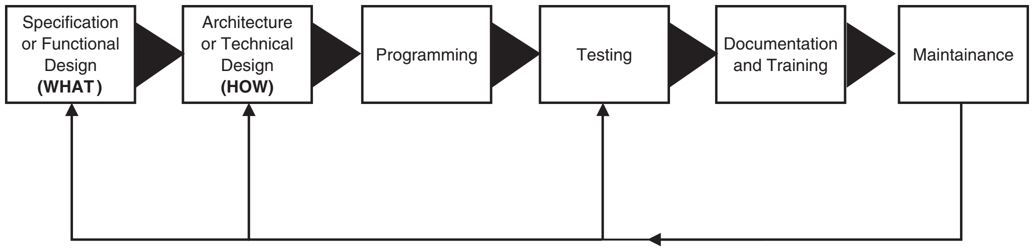
Specification or Functional Design (WHAT)
Architecture or Technical Design (HOW)
Programming
Testing
Documentation and Training
Maintainance

FIGURE 1.1
Essential Steps in the Traditional Enterprise Software Development Process

One response is that software is intrinsically more complex than hardware because it has more states, or modes of behavior. No machine has 1,000 operating modes, but any integrated enterprise business application system is likely to have 2,500 or more input forms. Software complexity is conventionally described as proportional to some factor—say, ***N***—

depending on the type of program, times the number of inputs, ***I***, multiplied by the number of outputs, ***O***, to some power, ***P***. Thus

$$\text{software complexity} = N^*I^*O^P$$

This can be thought of as increasing with the number of input parameters but growing exponentially with the number of output results.

Computers, controlled by software, naturally have more states—that is, they have larger performance envelopes than do other, essentially mechanical, systems. Thus, they are more complex.

Sidebar 1.1: Computer Complexity

When one of the authors of this book went from being an aircraft designer to a computer architect in 1967, he was confronted by the complexity of the then newly developing multiprocessor computer. At the time, Marshall McLuhan's book *Understanding Media* was a popular read. In it, this Canadian professor of English literature stated that a supersonic air transport plane is far simpler than a multiprocessor computer system. This was an amazing insight for a professor of English literature, but he was correct.

One of the authors of this book worked on the structural optimization of the *Concorde* and on a structural aspect of the swing-wing of the Boeing SST. In 1968 he was responsible for making the Univac 1108 function as a three-way multiprocessor. Every night at midnight he reported to the Univac test floor in Roseville, Minnesota, where he was assigned three 1108 mainframe computers. He connected the new multiprocessor CRT console he had designed and loaded a copy of the Exec 8 operating system modified for this new functionality. Ten times in a row the OS crashed at a different step of the bootstrap process. He began to wonder if this machine were a finite automaton after all. Of course it was, and the diverse halting points were a consequence of interrupt races, but he took much comfort from reading Marshall McLuhan. Today, highly parallel machines are commonplace in business, industry, and the scientific laboratory—and they are indeed far more complex than supersonic transport aircraft (none of which are still flying now that the *Concorde* has been taken out of service).

Although software engineering has become a popular subject of many books and is taught in many university computing curricula, we find the engineering/manufacturing metaphor to be a bit weak for software development. Most of a hardware product's potential problems become apparent in testing. Almost all of them can be corrected by tuning the hardware manufacturing process to reduce product and/or process variability. Software is different. Few potential problems can be detected in testing due to the complexity difference between software and hardware. None of them can be corrected by tuning the manufacturing process, because *software has no manufacturing process*! Making copies of eight CD-ROMs for shipment to the next customer along with a box of installation and user manuals offers little chance for fine-tuning and in any case introduces no variability. It is more like book publishing, in which you can at most slip an errata sheet into the misprinted book before shipping, or, in the case of software, an upgrade or fix-disk.

So, what is the solution? Our contention is that because errors in software are almost all created well upstream in the design process, and because software is all design and development, with no true manufacturing component, everything that can be done to create bug-free software must be done as far upstream in the design process as possible. Hence our advocacy of Taguchi Methods (see Chapters 2, 15, and 17) for robust software architecture. Software development is an immensely more taxing process than hardware development. Although there is no silver bullet, we contend that the Taguchi Methods described in the next chapter can be deployed as a key instrument in addressing software product quality upstream at the design stage. Processes are often described as having upstream activities such as design and downstream activities such as testing. This book advocates moving the quality-related aspects of development as far upstream in the development process as possible. The RSDM presented in this book provides a powerful framework to develop *trustworthy software* in a time- and cost-effective manner.

This introductory chapter is an overview of the software development situation today in the view of one of the authors. Although he has been developing both systems and applications software since 1957, no single individual's career can encompass the entire spectrum of software design and development possibilities. We have tried in this chapter to indicate when we are speaking from personal experience and sharing our personal opinions, and when we are referring to the experience of others.

Software Development Strategies and Life-Cycle Models

Here we will describe from a rather high altitude the various development methods and processes employed for software today. We focus on designing, creating, and maintaining large-scale enterprise application software, whether developed by vendors or in-house development teams. The creation and use of one-off and simple interface programs is no challenge. Developing huge operating systems such as Microsoft XP with millions of lines of code (LOC), or large, complex systems such as the FAA's Enroute System, bring very special problems of their own and are beyond the scope of this book. This is not to say that the methodology we propose for robust software architecture is not applicable; rather, we will not consider their applications here. The time-honored enterprise software development process generally follows these steps (as shown in Figure 1.1):

- Specification or functional design, done by system analysts in consort with the potential end users of the software to determine *why* to do this, *what* the application will do, and *for whom* it will do it.
- Architecture or technical design, done by system designers as the way to achieve the goals of the functional design using the computer systems available, or to be acquired, in the context of the enterprise as it now operates. This is *how* the system will function.
- Programming or implementation, done by computer programmers together with the system designers.
- Testing of new systems (or regression testing of modified systems) to ensure that the goals of the functional design and technical design are met.
- Documentation of the system, both intrinsically for its future maintainers, and extrinsically for its future users. For large systems this step may involve end-user training as well.
- Maintenance of the application system over its typical five-year life cycle, employing the design document now recrafted as the Technical Specification or System Maintenance Document.

This model and its variations, which we overview in this chapter, are largely software developer-focused rather than being truly customer-centric. They have traditionally attempted to address issues such as project cost and implementation overruns rather than customer satisfaction issues such as software reliability, dependability, availability, and upgradeability. It may also be pointed out that all these models follow the "design-test-design" approach. Quality assurance is thus based on fault detection rather than fault prevention, the central tenet of this book's approach. We will also discuss—in Chapters 2, 4, and 11 in particular—how the model that we propose takes a fault-prevention route that is based not only on customer specifications but also on meeting the totality of the user's needs and environment.

A software development model is an organized strategy for carrying out the steps in the life cycle of a software application program or system in a predictable, efficient, and repeatable way. Here we will begin with the primary time-honored models, of which there are many variants. These are the build-and-fix model, the waterfall model, the evolutionary model, the spiral model, and the iterative development model. Rapid prototyping and extreme programming are processes that have more recently augmented the waterfall model. The gradual acceptance of OOP over the past decade, together with its object frameworks and sophisticated integrated development environments, have been a boon to software developers and have encouraged new developments in automatic programming technology.

These life-cycle models and their many variations have been widely documented. So have current technology enhancements in various software development methods and process improvement models, such as the *Rational Unified Process (RUP)*, the *Capability Maturity Model (CMM),* and the *ISO 9000-3 Guidelines*. Therefore, we will consider them only briefly. We will illustrate some of the opportunities we want to address using the RSDM within the overall framework of DFTS technology. It is not our purpose to catalog and compare existing software development technology in any detail. We only want to establish a general context for introducing a new approach.

Build-and-Fix Model

The *build-and-fix* model was adopted from an earlier and simpler age of hardware product development. Those of us who bought early Volkswagen automobiles in the 1950s and '60s remember it well. As new models were brought out and old models updated, the cars were sold apparently without benefit of testing, only to be tested by the customer. In every case, the vehicles were promptly and cheerfully repaired by the dealer at no cost to their owners, except for the inconvenience and occasional risk of a breakdown. This method clearly works, but it depends on having a faithful and patient customer set almost totally dependent on the use of your product! It is the same with software. A few well-known vendors are famous for their numerous free upgrades and the rapid proliferation of new versions. This always works best in a monopolistic or semimonopolistic environment, in which the customer has limited access to alternative vendors. Unfortunately in the build-and-fix approach, the product's overall quality is never really addressed, even though some of the development issues are ultimately corrected. Also, there is no way to feed back to the design process any proactive improvement approaches. Corrections are put back into the market as bug fixes, service packs, or upgrades as soon as possible as a means of marketing "damage control." Thus, little learning takes place within the development process. Because of this, build-and-fix is totally reactive and, by today's standards, is not really a development model at all. However, the model shown in Figure 1.2 is perhaps still the approach most widely used by software developers today, as many will readily, and somewhat shamefully, admit.

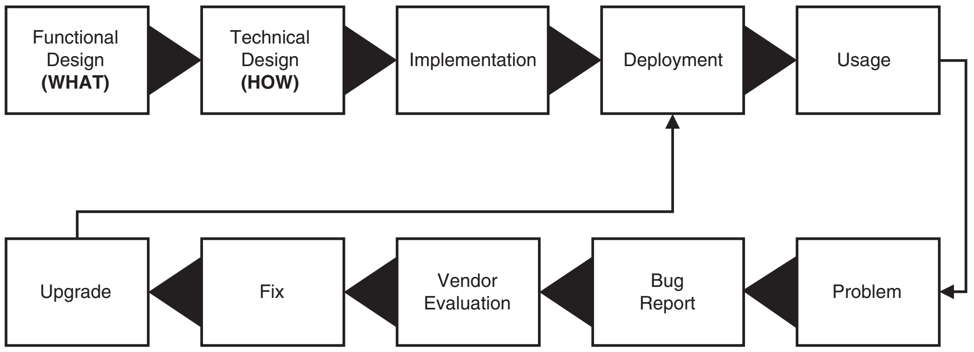

FIGURE 1.2
Build-and-Fix Software Development Model

Waterfall Model

The classic waterfall model was introduced in the 1970s by Win Royce at Lockheed. It is so named because it can be represented or graphically modeled as a cascade from establishing requirements, to design creation, to program implementation, to system test, to release to customer, as shown in Figure 1.3. It was a great step forward in software development as an engineering discipline. The figure also depicts the single-level feedback paths that were not part of the original model but that have been added to all subsequent improvements of the model; they are described here. The original waterfall model had little or no feedback between stages, just as water does not reverse or flow uphill in a cascade but is drawn ever downward by gravity. This method might work satisfactorily if design requirements could be perfectly addressed before flowing down to design creation, and if the design were perfect when program implementation began, and if the code were perfect before testing began, and if testing guaranteed that no bugs remained in the code before the users applied it, and of course if the users never changed their minds about requirements. Alas, none of these things is ever true. Some simple hardware products may be designed and manufactured this way, but this model has been unsatisfactory for software products because of the complexity issue. It is simply impossible to guarantee correctness of any program of more than about 169 lines of code by any process as rigorous as mathematical proof. Proving program functionality *a priori* was advantageous and useful in the early days of embedded computer control systems, when such programs were tiny, but today's multifunction cell phones may require a million lines of code or more!

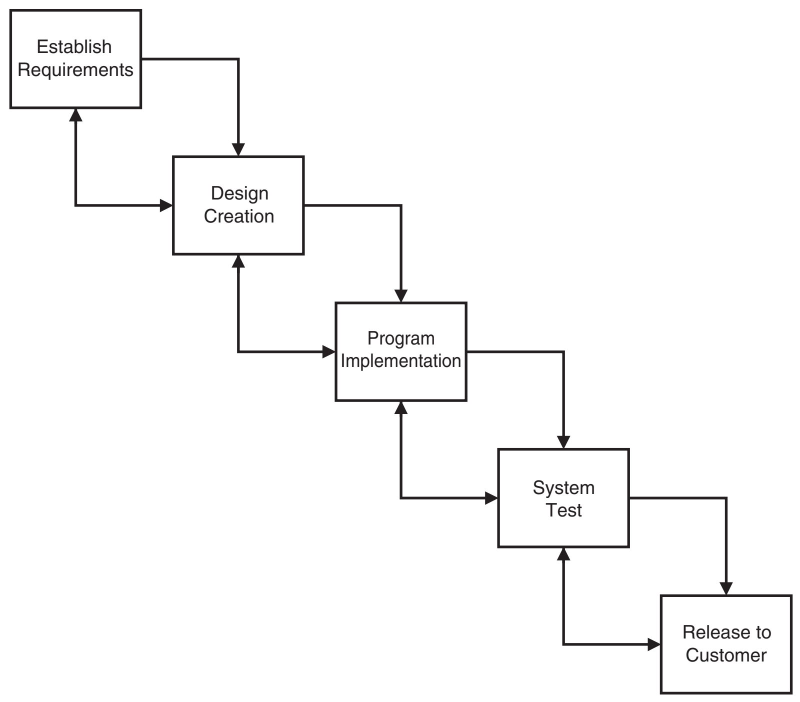

FIGURE 1.3
Waterfall Model for Software Development

Rapid Prototyping Model

Rapid prototyping has long been used in the development of one-off programs, based on the familiar model of the chemical engineer's pilot plant. More recently it has been used to prototype larger systems in two variants—the "throwaway" model and the "operational" model, which is really the incremental model to be discussed later. This development process produces a program that performs some essential or perhaps typical set of functions for the final product. A throwaway prototype approach is often used if the goal is to test the implementation method, language, or end-user acceptability. If this technology is completely viable, the prototype may become the basis of the final product development, but normally it is merely a vehicle to arrive at a completely secure functional specification, as shown in Figure 1.4. From that point on the process is very similar to the waterfall model. The major difference between this and the waterfall model is not just the creation of the operational prototype or functional subset; the essence is that it be done very quickly—hence the term *rapid* prototyping.[3]

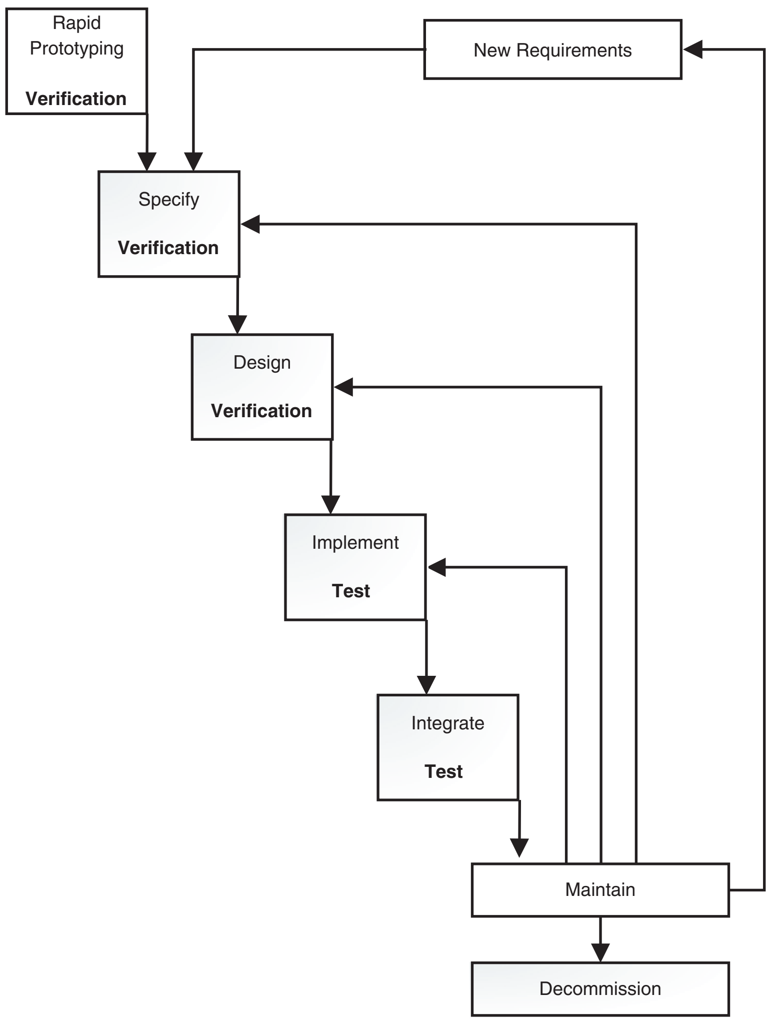

FIGURE 1.4
Rapid Prototyping Model

Incremental Model

The incremental model recognizes that software development steps are not discrete. Instead, Build 0 (a prototype) is improved and functionality is added until it becomes Build 1, which becomes Build 2, and so on. These builds are not the versions released to the public but are merely staged compilations of the developing system at a new level of functionality or completeness. As a major system nears completion, the project manager may schedule a new build every day at 5 p.m. Heaven help the programmer or team who does not have their module ready for the build or whose module causes compilation or regression testing to fail! As Figure 1.5 shows, the incremental model is a variant of the waterfall and rapid prototyping models. It is intended to deliver an operational-quality system at each build stage, but it does not yet complete the functional specification.[4] One of the biggest advantages of the incremental model is that it is flexible enough to respond to critical specification changes as development progresses. Another clear advantage is that analysts and developers can tackle smaller chunks of complexity. Psychologists teach the "rule of seven": the mind can think about only seven related things at once. Even the trained mind can juggle only so many details at once. Users and developers both learn from a new system's development process, and any model that allows them to incorporate this learning into the product is advantageous. The downside risk is, of course, that learning exceeds productivity and the development project becomes a research project exceeding time and budget or, worse, never delivers the product at all. Since almost every program to be developed is one that has never been written before, or hasn't been written by this particular team, research program syndrome occurs all too often. However, learning need not exceed productivity if the development team remains cognizant of risk and focused on customer requirements.

Extreme Programming

Extreme Programming (XP) is a rather recent development of the incremental model that puts the client in the driver's seat. Each feature or feature set of the final product envisioned by the client and the development team is individually scoped for cost and development time. The client then selects features that will be included in the next build (again, a build is an operational system at some level of functionally) based on a cost-benefit analysis. The major advantage of this approach for small to medium-size systems (10 to 100 man-years of effort) is that it works when the client's requirements are vague or continually change. This development model is distinguished by its flexibility because it can work in the face of a high degree of specification ambiguity on the user's part. As shown in Figure 1.6, this model is akin to repeated rapid prototyping, in which the goal is to get certain functionality in place for critical business reasons by a certain time and at a known cost.[5]

02_0131872508_ch01.qxd 8/9/06 4:43 PM Page 15

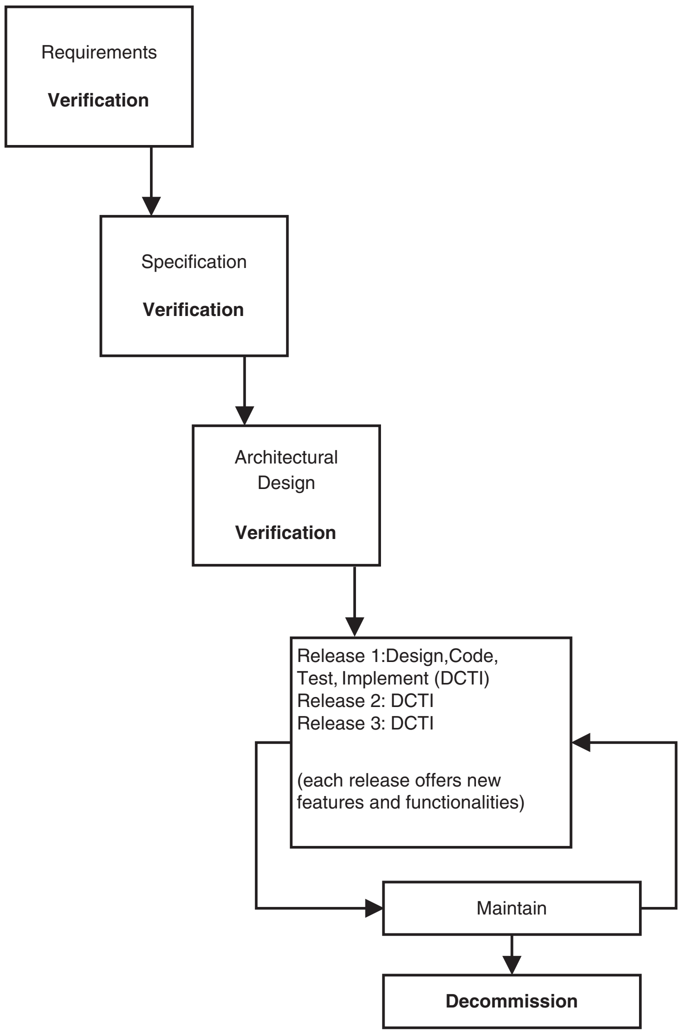

FIGURE 1.5
Incremental Model

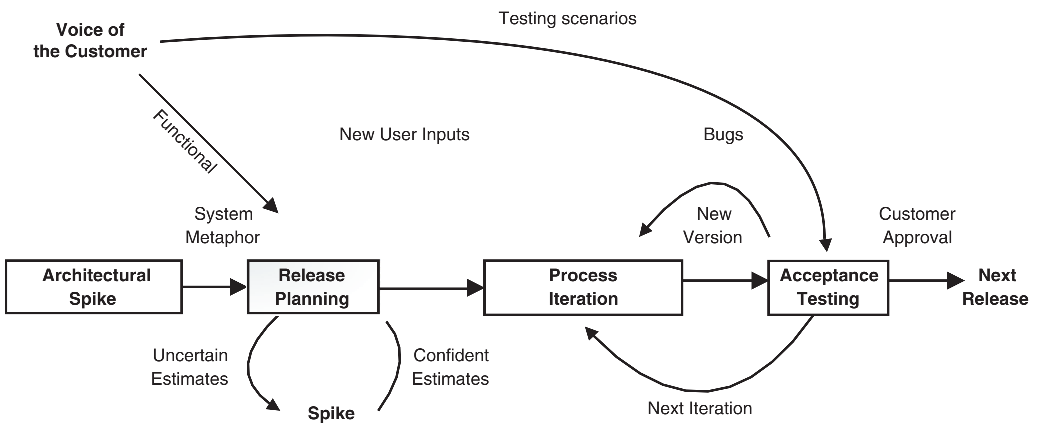

FIGURE 1.6
Extreme Programming Model

Adapted from Don Wells: www.extremeprogramming.org. Don Wells XP website gives an excellent overview of the XP development process. A more exhaustive treatment is given in Kent Beck. *Extreme Programming Explained* (Boston: Addison-Wesley, 2000)

Spiral Model

The spiral model, developed by Dr. Barry Boehm[6] at TRW, is an enhancement of the waterfall/rapid prototype model, with risk analysis preceding each phase of the cascade. You can imagine the rapid prototyping model drawn in the form of a spiral, as shown in Figure 1.7. This model has been successfully used for the internal development of large systems and is especially useful when software reuse is a goal and when specific quality objectives can be incorporated. It does depend on being able to accurately assess risks during development. This depends on controlling all factors and eliminating or at least minimizing exogenous influences. Like the other extensions of and improvements to the waterfall model, it adds feedback to earlier stages. This model has seen service in the development of major programming projects over a number of years, and is well documented in publications by Boehm and others.

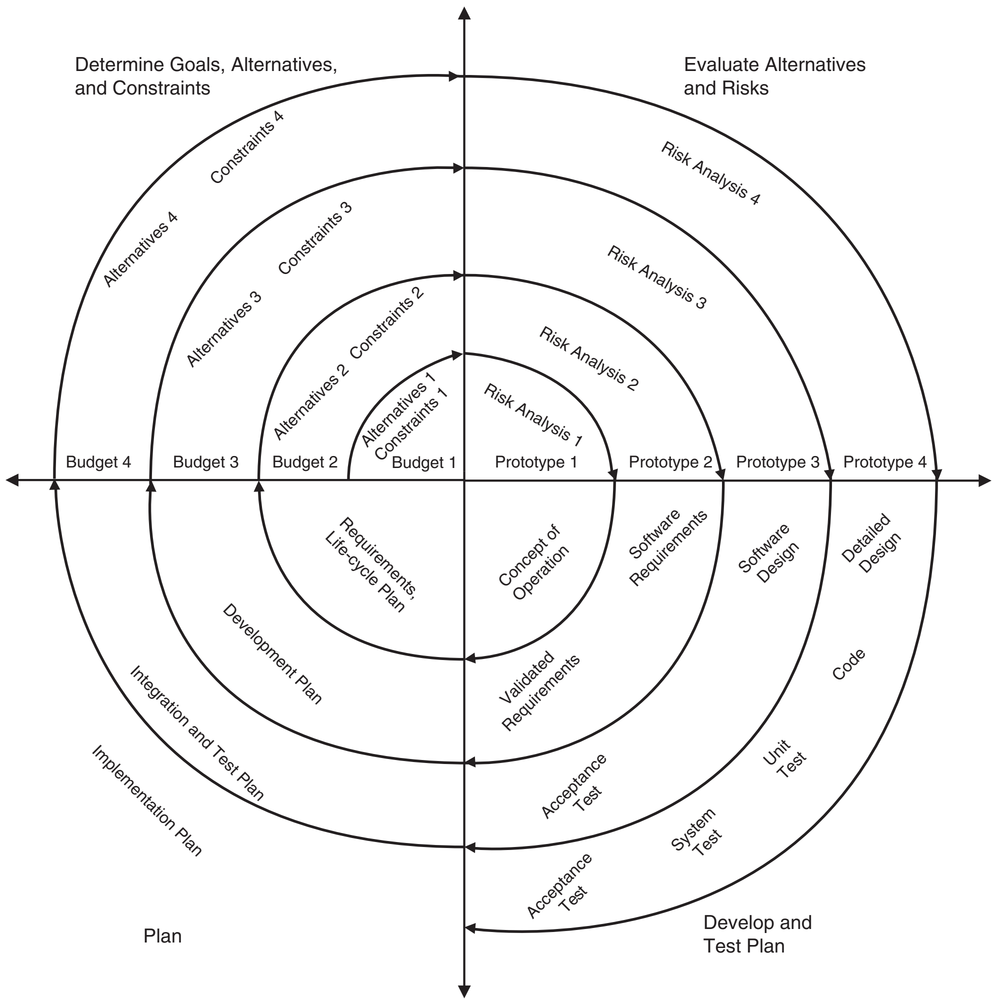

FIGURE 1.7
Spiral Model

Adapted from B. W. Boehm, “A Spiral Model of Software Development and Enhancement,” *IEEE Computer*, 21 (May 1988), pp. 61–72.

Object-Oriented Programming

Object-Oriented Programming (OOP) technology is not a software development model. It is a new way of designing, writing, and documenting programs that came about after the development of early OOP languages such as C++ and Smalltalk. However, OOP does enhance the effectiveness of earlier software development models intended for procedural programming languages, because it allows the development of applications by slices rather than by layers. The central ideas of OOP are encapsulation and polymorphism, which dramatically reduce complexity and increase program reusability. We will give examples of these from our experience in later chapters. OOP has become a major development technology, especially since the wide acceptance of the Java programming language and Internet-based application programs. OOP analysis, design, and programming factor system functionality into objects, which include data and methods designed to achieve a specific, scope-limited set of tasks. The objects are implementations or instances of program classes, which are arranged into class hierarchies in which subclasses inherit properties (data and methods) from superclasses. The OOP model is well supported by both program development environments (PDEs) and more sophisticated team-oriented integrated development environments (IDEs), which encourage or at least enable automatic code generation.

OOP is a different style of programming than traditional procedural programming. Hence, it has given rise to a whole family of software development models. Here we will describe the popular Booch Round-Tripping model,[7] as shown in Figure 1.8. This model assumes a pair of coordinated tool sets—one for analysis and design and another for program development. For example, you can use the Uniform Modeling Language (UML) to graphically describe an application program or system as a class hierarchy. The UML can be fed to the IDE to produce a Java or C++ program, which consists of the housekeeping and control logic and a large number of stubs and skeleton programs. The various stub and skeleton programs can be coded to a greater or lesser extent to develop the program to a given level or "slice" of functionality. The code can be fed back or "round-tripped" to the UML processor to create a new graphical description of the system. Changes and additions can be made to the new UML description and a new program generated. This general process is not really new. The Texas Instruments TEF tool set and the Xcellerator tool set both allowed this same process with procedural COBOL programs. These tools proved their worth in the preparation for the Y2K crisis. A working COBOL application with two-digit year dates could be reverse-engineered to produce an accurate flowchart of the application (not as it was originally programmed, but as it was actually implemented and running). Then it could be modified at a high level to add four-digit year date capability. Finally, a new COBOL program could be generated, compiled, and tested. This older one-time reverse engineering is now built into the design feedback loop of the Booch Round-Trip OOP development model. It can be further supported with code generators that can create large amounts of code based on recurring design patterns.

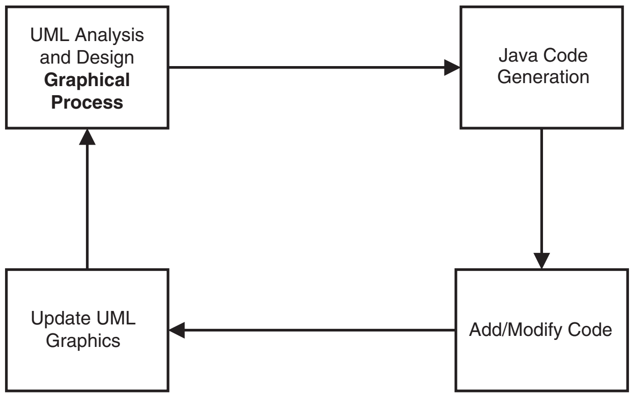

FIGURE 1.8
Round-Tripping Model

Iterative Development or Evolutionary Model

The iterative development model is the most realistic of the traditional software development models. Rather than being open-loop like build-and-fix or the original waterfall models, it has continuous feedback between each stage and the prior one. Occasionally it has feedback across several stages in well-developed versions, as illustrated in Figure 1.9. In its most effective applications, this model is used in an *incremental iterative* way. That is, applying feedback from the last stage back to the first stage results in each iteration's producing a useable executable release of the software product. A lower feedback arrow indicates this feature, but the combined incremental iterative method schema is often drawn as a circle. It has been applied to both procedural and object-oriented program development.

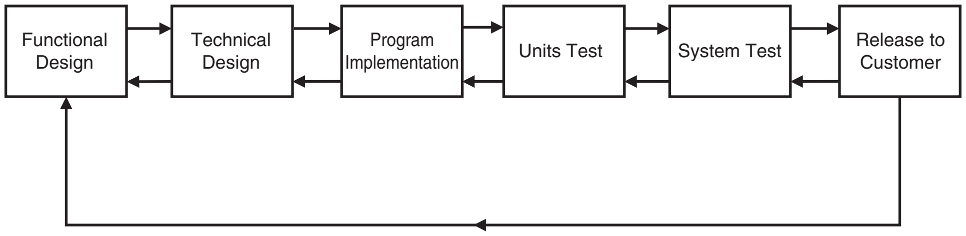

FIGURE 1.9
Iterative Model of Software Development

Comparison of Various Life-Cycle Models

Table 1.1 is a high-level comparison between software development models that we have gathered into groups or categories. Most are versions or enhancements of the waterfall model. The fundamental difference between the models is the amount of engineering documentation generated and used. Thus, a more "engineering-oriented" approach may have higher overhead but can support the development of larger systems with less risk and can support complex systems with long life cycles that include maintenance and extension requirements.

TABLE 1.1
Comparison of Traditional Software Development Models

Model	Pros	Cons
Build-and-fix	OK for small one-off programs	Useless for large programs
Waterfall	Disciplined, document-driven	Result may not satisfy client
Rapid prototyping	Guarantees client satisfaction	May not work for large applications
Extreme programming	Early return on software development	Has not yet been widely used
Spiral	Ultimate waterfall model	Large system in-house development only
Incremental	Promotes maintainability	Can degenerate to build-and-fix
OOP	Supported by IDE tools	May lack discipline
Iterative	Can be used by OOP	May allow overiteration

Software Process Improvement

Although the legacy models for software development just discussed are honored by time and are used extensively even today, they are surely not the latest thinking on this subject. We will describe only briefly RUP, CMM, and ISO 9000 software process improvement development models, because they will receive attention in later chapters. These are very different things but are considered here as a diverse set of technologies that are often "compared" by software development managers. RUP and CMM are the result of considerable government-sponsored academic research and industrial development. When rigorously applied, they yield good, even excellent, results. They also provide a documentation trail that eases the repair of any errors and bugs that do manage to slip through a tightly crafted process net. These newer methods are widely used by military and aerospace contractors who are required to build highly secure and reliable software for aircraft, naval vessels, and weapons systems. In our experience they have had relatively little impact on enterprise software development so far, whether internally or by way of third-party vendors.

Rational Unified Process

The Rational Unified Process (RUP) is modeled in two dimensions, rather than linearly or even circularly, as the previously described models are. The horizontal axis of Table 1.2 represents time, and the vertical axis represents logical groupings of core activities.[8]

TABLE 1.2
A Two-Dimensional Process Structure—Rational Unified Model

	Phase			
Workflow	Inception	Elaboration	Construction	Transition to Next Phase
Application model	Definition	Comparison	Clarification	Consensus
Requirements	Gathering	Evaluation	User review	Approval
Architecture	Analysis	Design	Implementation	Documentation
Test	Planning	Units test	System test	Regression testing
Deployment	User training	User planning	Site installation	User regression testing
Configuration management	Long-range planning	Change management	Detailed plan for evolution	Planning approvals
Project management	Statements of work	Contractor or team identification	Bidding and selection	Let contracts or budget internal teams
Environment	Hiring or relocation	Team building	Training	Certification

The Rational Model is characterized by a set of software best practices and the extensive application of use cases. A use case is a set of specified action sequences, including variant and error sequences, that a system or subsystem can perform interacting with outside actors.[9] The use cases are very effective at defining software functionality[10] and even planning to accommodate error or "noise." However, the RUP's most important advantage is its iterative process that allows changes in functional requirements also to be accommodated as they inevitably change during system development. Not only do external circumstances reflect changes to the design, but also the user's understanding of system functionality becomes clearer as that functionality develops. The RUP has been developing since 1995 and can claim well over 1,000 user organizations.

Capability Maturity Model

The Capability Maturity Model (CMM) for software development was developed by the Software Engineering Institute at Carnegie Mellon University. CMM is an organizational maturity model, not a specific technology model. Maturity involves continuous process improvement based on evaluation of iterative execution, gathering results, and analyzing metrics. As such, it has a very broad universe of application. The CMM is based on four principles:[11]

- Evolution (process improvement) is possible but takes time. The process view tells us that a process can be incrementally improved until the result of that process becomes adequately reliable.
- Process maturity has distinguishable stages. The five levels of the CMM are indicators of process maturity and capability and have proven effective for measuring process improvement.
- Evolution implies that some things must be done before others. Experience with CMM since 1987 has shown that organizations grow in maturity and capability in predictable ways.
- Maturity will erode unless it is sustained. Lasting changes require continued effort.

The five levels of the CMM, in order of developing maturity, are as follows:

- **Level 1** (Ad Hoc): Characterized by the development of software that works, even though no one really understands why. The team cannot reliably repeat past successes.
- **Level 2** (Repeatable): Characterized by requirements management, project planning, project tracking, quality assurance, configuration management.
- **Level 3** (Defined): Organization project focus and project definition, training program, integrated software management, software product engineering, intergroup coordination, peer reviews.

- **Level 4** (Managed): Quantitative process management, software quality management.
- **Level 5** (Optimizing): Defect prevention, technology change management, process change management.

Note that level 3 already seems to be higher than most software development organizations attain to, and would seem to be a very worthy goal for any development organization. However, the CMM has two levels of evolutionary competence/capability maturity above even this high-water mark. CMM as well as Capability Maturity Model Integration (CMMI) and PCMM (People Capability Maturity Model) have had enthusiastic acceptance among software developers in India. In 2000, the CMM was upgraded to CMMI. The Software Engineering Institute (SEI) no longer maintains the CMM model. IT firms in India accounted for 50 out of 74 CMM level 5-rated companies worldwide in 2003.[12] They are also leading in other quality management systems, such as Six Sigma, ISO 9001, ISO 14001, and BS 7799. It would seem that embracing a multitude of systems and models has helped software developers in India take a rapid lead in product and process improvement, *but still there is no silver bullet!*

ISO 9000-3 Software Development Guidance Standard

This guidance standard is a guideline for the application of standards to the development, supply, and maintenance of computer software. It is not a development model like RUP or even a organization developmental model like CMM. Neither is it a certification process. It is a guidance document that explains how ISO 9001 should be interpreted within the software industry (see Figure 1.10). It has been used since 1994, having been introduced as ISO 9001 Software Quality Management.[13] It was updated in 2002 as ISO 9000-3. Prudent compliance of ISO 9000-3 may result in the following benefits:

- Increases the likelihood of quality software products
- Gives you a competitive advantage over non-ISO 9000 certified development vendors
- Assures customers of the end product's quality
- Defines the phases, roles, and responsibilities of the software development process
- Measures the efficiency of the development process
- Gives structure to what is often a chaotic process

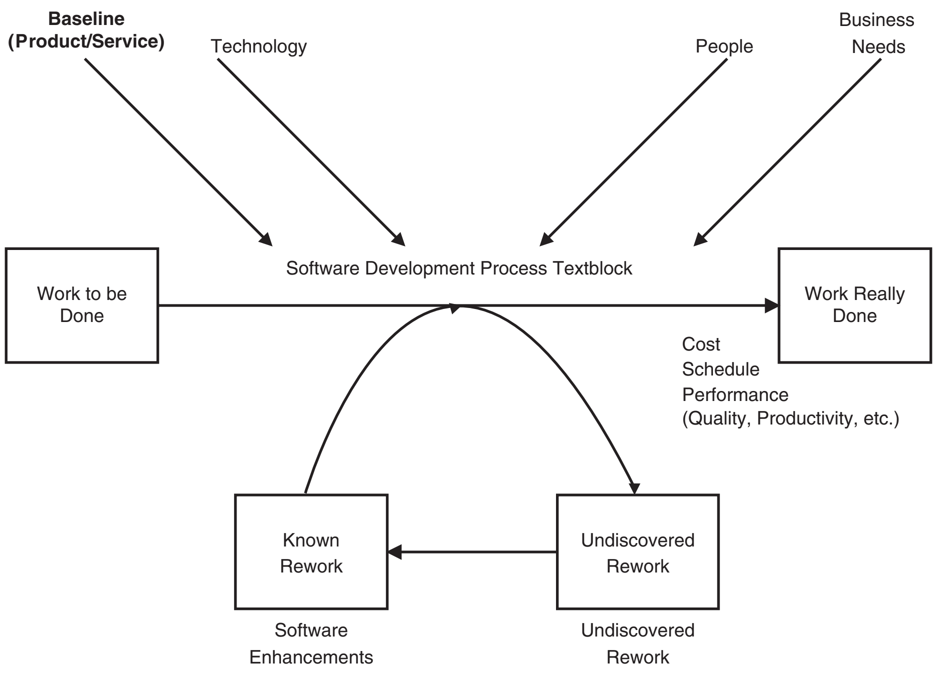

FIGURE 1.10
ISO 9000-3 Software Development Model

The document was designed as a checklist for the development, supply, and maintenance of software. It is not intended as a certification document, like other standards in the ISO 9000 series. Copies of the guideline can be ordered from the ISO in Switzerland. Also, many consulting firms have Web sites that present the ISO 9000-3 guidelines in a cogent, simplified, and accessible way.[14]

The Tickit process was created by the British Computer Society and the United Kingdom Department of Trade and Industry for actually certifying ISO 9000-3 software development.[15] This partnership has turned the ISO 9000-3 guideline standard into a compliance standard. It allows software vendors to be certified for upholding the ISO 9000-3 standard after passing the required audits. As with other ISO 9000 standards, there is a great deal of emphasis on management, organization, and process that we will not describe in this brief overview. Rather, we will emphasize the ISO development procedures that control software design and development. These include the use of life-cycle models to organize and create a suitable design method by reviewing past designs and considering what is appropriate for each new project. The following three sets of issues are addressed:

- Preparation of a software development plan to control:
    - Technical activities (design, coding, testing)
    - Managerial activities (supervision, review)
    - Design input (functional specs, customer needs)
    - Design output (design specs, procedures)
    - Design validation
    - Design verification
    - Design review
    - Design changes
- Development of procedures to control the following documents and data:
    - Specifications
    - Requirements
    - Communications
    - Descriptions
    - Procedures
    - Contracts
- Development of procedures to plan, monitor, and control the production, installation, and service processes for managing the following:
    - Software replication
    - Software release
    - Software installation
    - Develop software test plans (for unit and integration testing)
    - Perform software validation tests
    - Document testing procedures

Much of this sounds like common sense, and of course it is. The advantage of incorporating such best practices and conventional wisdom into a guidance standard is to encourage uniformity among software vendors worldwide and leveling of software buyers' expectations so that they are comfortable with purchasing and mixing certified vendors' products.

Comparison of RUP, CMM, and ISO 9000

A brief comparison of these process improvement systems is provided in Table 1.3. Such a comparison is a bit awkward, like comparing apples and oranges, but apples and oranges are both fruit. In our experience, software development managers often ask each other, "Are you using RUP, CMM, or ISO 9000?" as if these were logically discrete alternatives, whereas they are three different things.

TABLE 1.3
Comparison of RUP, CMM, and ISO 9000

Method	Pros	Cons
RUP	Well supported by tools Supports OOP development More than 1,000 users	Expensive to maintain High training costs Used downstream with RSDM
CMM	Sets very high goals Easy to initiate Hundreds of users	Completely process-oriented Requires long-term top management support
ISO 9000-3	Provides process guidelines Documentation facilitated Comprehensive, detailed	Some firms may seek to gain certification without process redesign

The RUP is very well supported by an extensive array of software development and process management tools. It supports the development of object-oriented programs. It is expensive to install and has a rather steep learning curve with high training costs but is well worth the time and cost to implement. RUP is estimated to be in use by well over 1,000 firms. Its usability with RSDM will be detailed later. The CMM sets very high ultimate goals but is easy to initiate. However, it does require a long-term commitment from top management to be effective over time and to be able to progress to maturity level 3 and beyond. It is estimated to have well over 400 users in the United States. As stated earlier, it is very popular in India, where the majority of CMM user firms are located. ISO 9000-3 was updated in 2002. It is essential for the development of third-party enterprise software to be sold and used in the EEC. A large number of consulting firms in both Europe and North America are dedicated to training, auditing, and compliance coaching for ISO 9000. Users report that it works quite well, although at first it appears to be merely institutionalized common sense. Perhaps the only downside is, because it is a required certification, some firms may just try to get the certification without really redesigning software development processes to conform to the guidelines.

Table 21.4 in Chapter 21 compares different quality systems currently common in software companies. These systems serve different needs and can coexist. The need for integration is discussed in Chapter 21 (see Case Study 21.1) and Chapter 27.

ADR Method

ADR stands for assembly (A), disassembly (D), and reassembly (R)—the major aspects of component-based software development.[16] Software components in enterprise systems are fairly large functional units that manage the creation and processing of a *form*, which usually corresponds to an actual business form in its electronic instance. For example, a general ledger (GL) system may consist of 170 components, some 12 or more of which must be used to create a general ledger for a firm from scratch. Each component in the GL application corresponds to an *accounting* function that the application is designed to perform. This approach arose in the early days of 4GL (Fourth-Generation Language) software development and has continued to be popular into the OOP era. OOP components tend to be somewhat smaller than 4GL components due to the class factoring process that naturally accompanies Object-Oriented Analysis and Design. In the cited paper,[16] Professor Ramamoorthy describes the evolution of software quality models and generalizes and classifies them.

Seven Components of the Robust Software Development Process

Software has become an increasingly indispensable element of a wide range of military, industrial, and business applications. But it is often characterized by high costs, low reliability, and unacceptable delays. Often, they are downright unacceptable (see Sidebar 1.2). Software life-cycle costs (LCC) typically far exceed the hardware costs. Low software quality has a direct impact on cost. Some 40% of software development cost is spent testing to remove errors and to ensure high quality, and 80 to 90% of the software LCC goes to fix, adapt, and expand the delivered program to meet users' unanticipated, changing, or growing needs.[17] While the software costs far exceed hardware costs, the corresponding frequency of failure rate between software and hardware could be as high as 100:1. This ratio can be even higher for more advanced microprocessor-based systems.[18] Clearly, these are huge issues that cannot be addressed effectively by continuing to deploy traditional software development approaches.

It is well known that various quality issues are interrelated. Moreover, high costs and delays can be attributed to low software reliability.[18] Thus, it is conceivable that several objectives may be met with the correct strategic intervention. Quality has a great many useful attributes, and you must clearly understand the customer perspectives throughout the software life cycle. This helps you not only understand the changing user needs but also avoid cost escalation, delays, and unnecessary complexity. You need to deploy a multipronged strategy to address quality issues in large and complex software such as enterprise applications. The Seven Components of a Robust Software Development Process are shown in Figure 1.11. They are as follows:

1. A steadfast development process that can provide interaction with users by identifying their spoken and unspoken requirements throughout the software life cycle.
2. Provision for feedback and iteration between two or more development stages as needed.
3. An instrument to optimize design for reliability (or other attributes), cost, and cycle time at once at upstream stages. This particular activity, which addresses software product robustness, is one of the unique features of the RSDM, because other software development models do not.
4. Opportunity for the early return on investment that incremental development methods provide.
5. Step-wise development to build an application as needed and to provide adequate documentation.
6. Provision for risk analyses at various stages.
7. Capability to provide for object-oriented development.

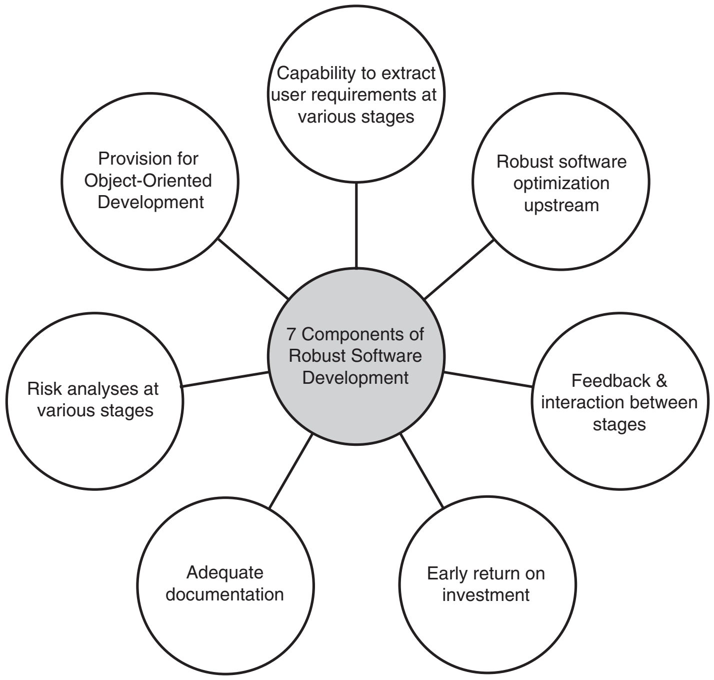
Capability to extract user requirements at various stages
Provision for Object-Oriented Development
Robust software optimization upstream
7 Components of Robust Software Development
Risk analyses at various stages
Feedback & interaction between stages
Adequate documentation
Early return on investment

FIGURE 1.11
Seven Components of the Robust Software Development Process

Robust Software Development Model

Our proposed model for software development is based on DFTS technology, as shown in Figure 2.6 in Chapter 2. DFTS technology consists of Robust Software Development Model, Software Design Optimization Engineering, and Object-Oriented Design Technology. As you will soon see, it is a more elaborate combined form of the cascade and iterative models with feedback at every level. In fact, it attempts to incorporate the best practices and features from various development methodologies and collectively provides for a customer-focused robust software technology. It is intended to meet all seven key requirements for a robust software architecture development method just identified. Although Taguchi Methods have been applied to upstream software design in a few cases,[19, 20] there is not yet an extensive body of literature devoted to this area.

The primary focus of this book is to explain this model in the context of robust software design and to show you how you can use it for DFTS. The purpose of this book is to give you a map for robust software design from the hardware design arena to that of software design and development. We will also establish a context for methodologies such as Taguchi Methods and Quality Function Deployment (QFD) in the software arena. We will show you how they can be used as the upstream architectural design process for some of the established software quality models using Professor Ramamoorthy's taxonomy, as well as the software quality management processes that will allow the development organization using it to become a learning organization.

Sidebar 1.2: Mission-Critical Aircraft Control Software

The control computer of a Malaysian Airlines Boeing 777 seemed intent on crashing itself on a trip from Perth to Kuala Lumpur on August 1, 2005. According to *The Australian* newspaper, the Malaysian flight crew had to battle for control of the aircraft after a glitch occurred in the computerized control system. The plane was about an hour into the flight when it suddenly climbed 3,000 feet and almost stalled. The Australian Air Transport Safety Bureau report posted on its Web site said the pilot was able to disconnect the autopilot and lower the nose to prevent the stall, but the auto throttles refused to disengage. When the nose pitched down, they increased power.[a] Even pushing the throttles to idle didn't deter the silicon brains, and the plane pitched up again and climbed 2,000 feet the second time. The pilot flew back to Perth on manual, but the auto throttles wouldn't turn off. As he was landing, the primary flight display gave a false low airspeed warning, and the throttles jammed again. The display also warned of a nonexistent wind shear. Boeing spokesman Ken Morton said it was the only such problem ever experienced on the 777, but airlines have been told via an emergency directive to load an earlier software version just in case. The investigation is focusing on the air data

---

[a]http://www.atsb.gov.au/aviation/occurs/occurs_detail.cfm?ID=767

inertial data reference unit, which apparently supplied false acceleration figures to the primary flight computer.

More recently, a JetBlue Airbus 320 flight from Burbank, California to New York on September 21, 2005 attracted several hours of news coverage when the control software locked its front landing gear wheels at a 90-degree angle at takeoff. After dumping fuel for three hours, the plane landed without injuries at LAX. However, the front landing gear was destroyed in the process in a blaze of sparks and fire. An NTSB official called the problem common[b]. A Canadian study issued last year reported 67 nose wheel incidents with Airbus 319, 320, and 321 models. The NTSB official leading the investigation said that "If we find a pattern, we will certainly do something." (From the *Los Angeles Times*, September 22, 2005)

Software failures in aircraft control systems are likely to incur a much higher social and economic cost than an error in a client's invoice, or even an inventory mistake. Unfortunately they are much harder to find and correct as well.

[b]http://www.airweb.faa.gov/Regulatory_and_Guidance_Library/rgad.nsf/0/25F9233FE09B613F8625706C005D0C53?OpenDocument

Key Points

- In spite of 50 years of software development methodology and process improvement, we need a new paradigm to develop increasingly complex software systems.
- Productivity gains in software development have not kept up with the performance increases in hardware. New hardware technology enables and encourages new applications, which require much larger and more complex programs.
- Perhaps a dozen models of software development aim to improve development productivity and/or enhance quality. They all work reasonably well when faithfully and diligently applied.
- The Department of Defense has sponsored a number of software development process improvement initiatives as a leader in the use of sophisticated computer applications and dedicated or embedded applications.
- The Design for Trustworthy Software (DFTS) technology addresses challenges of producing trustworthy software using a combination of the iterative Robust Software Development Model, Software Design Optimization Engineering, and Object-Oriented Design Technology.

Additional Resources

http://www.prenhallprofessional.com/title/0131872508

http://www.agilenty.com/publications

Internet Exercises

1. Search the Internet for U.S., Canadian, and Australian government reports on failure in aircraft control software. Is this problem getting better or worse?
2. Search the Internet for sites dedicated to the Rational Unified Process. How would you present an argument to your management to employ this process for software development in your own organization?
3. Look at the CMM site at the Software Development Institute. Can you see how this complex model could be applied in your organization?
4. What is the current status of the ISO 9000-3 Software Development Model, and what firms are supporting its use by software developers?

Review Questions

1. The CEO of the company for which you are MIS director asks why the new enterprise software for which he paid millions still has bugs. What do you tell him?
2. Which software development model does your organization or an organization you are familiar with employ? Do you consider it successful? If not, what does it lack, and where does it fail?
3. If a computer program is algorithmically similar to a mathematical theorem, why can't the person who designed it prove it will work properly before it is run?
4. How is object-oriented programming fundamentally different from earlier procedural programming technology? What promise do these differences hold for future software trustworthiness?

Discussion Questions and Projects

1. Create a table that shows the benefits and costs of using RUP, CMM, and ISO 9000-3 development models in terms of the size of the programs an organization develops and the number it completes per year.
2. Does your organization's software development process fit into the table you just created? If so, estimate the one-time costs and continuing costs of introducing the new model. Also estimate the benefits, long-term cost savings, and competitive advantages of using it.

Endnotes

[1]P. C. Patton, "The Development of the Idea of Computer Programming," QMCS White Paper 2003.3, St. Thomas University, June 2003, p. 4.

[2]Ibid, p. 6.

[3]S. R. Schach, *Object-Oriented and Classical Software Engineering* (Boston: McGraw-Hill, 2002), p. 71.

[4]Ibid, p. 73.

[5]K. Beck, "Embracing Change with Extreme Programming," *IEEE Computer*, 32 (October 1999), pp. 70–77.

[6]B. W. Boehm, "A Spiral Model of Software Development and Enhancement," *IEEE Computer*, 21 (May 1988), pp. 61–72.

[7]G. Booch, *Object-Oriented Analysis and Design with Applications*, 2nd Ed. (Menlo Park, CA: Addison-Wesley, 1994).

[8]P. Krutchen, *The Rational Unified Process* (Boston: Addison-Wesley, 2000), p. 23.

[9]J. Rumbaugh, I. Jacobson, G. Booch, *The Uniform Modeling Language* (Boston: Addison-Wesley, 1998).

[10]A. Cockburn, *Writing Effective Use Cases* (Boston: Addison-Wesley, 2001).

[11]K. M. Dymond, *A Guide to the CMM* (Annapolis, MD: Process Transition International, 2002), pp. 1–4.

[12]http://www.nasscom.org/artdisplay.asp?Art_id=3851

[13]M. G. Jenner, *Software Quality Management and ISO 9001* (New York: Wiley, 1995).

[14]http://www.praxiom.com/

[15]M. Callahan, *The Application of ISO 9000 to Software*, Team 7, 2002.

[16]C. V. Ramamoorthy, *Evolution and Evaluation of Software Quality Models*, Proceedings. 14th International Conference on Tools with Artificial Intelligence (ICTAI '02), 2002.

[17]W. Kuo, V. Rajendra Prasad, F. A. Tillman, Ching-Lai Wang. *Optimal Reliability Design* (Cambridge: Cambridge University Press, 2001), p. 5.

[18]D. Simmons, N. Ellis, H. W. Kuo. *Software Measurement: A Visualization Toolkit for Project Control and Process Improvement* (Englewood, NJ: Prentice-Hall, 1998).

[19]B. Kanchana, *Software Quality and Dependability Issues for the Airborne Surveillance Platform*, Doctoral Dissertation, Indian Institute of Science, Bangalore, India, Dec. 1998.

[20]G. Taguchi, S. Chowdhury, Yuin Wu, *Taguchi's Quality Engineering Handbook* (Boston: Jossey-Bass, 2004).