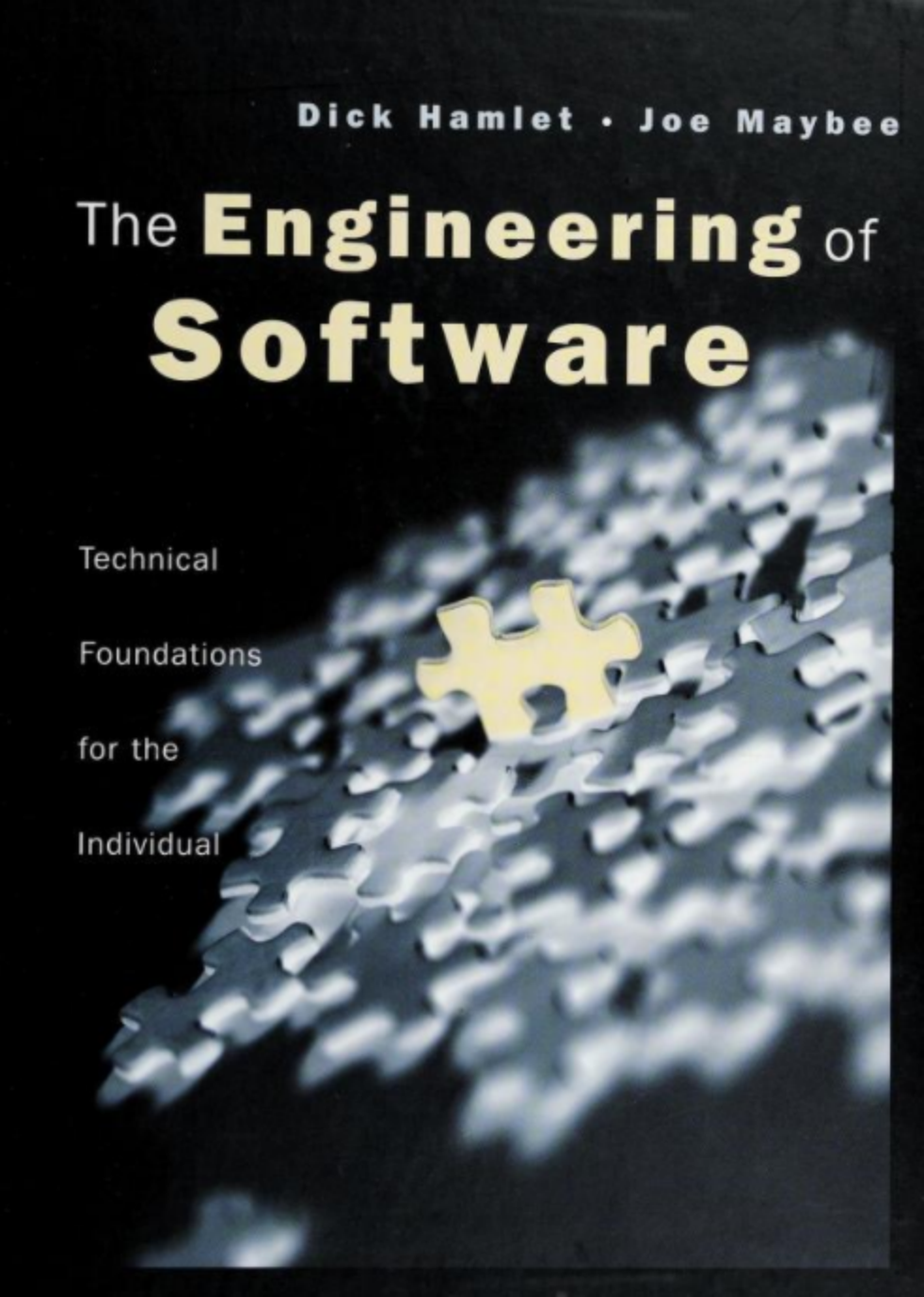
Dick Hamlet • Joe Maybee

The Engineering of Software



Technical Foundations for the Individual

Contents

chapter 1

Concepts of Software Development

Almost every printed article on computing, in newspapers, popular magazines, even in professional and technical literature, begins with a statement of how important and ubiquitous software has become. Some of these articles go on to stress the potential dangers of putting so many eggs in the software basket. And indeed software is increasingly entering our lives, with great potential benefits, and corresponding great risks. But software is not a phenomenon that occurs naturally in the world. People make software, as they make other artifacts like houses (although to be sure not in quite the same way). In making software, the people can do a good job or a poor one, and the resulting software will directly reflect their care and skill. "Software engineering" is a name for the collection of skills, ideas, and procedures intended to help people do a good job. To emphasize that our subject in this book is the technical skills for an individual, we have turned the name around into "the engineering of software."

1.1 Separation of Concerns

Software engineering is built upon one of the basic ideas of rational thought, an idea that is almost too obvious to write down, yet profoundly useful and important. Here it is:

> In thinking about something too complicated to master, don't try to do it all at once. Break it down into less complicated parts, and concentrate on each one in turn.

Any kind of a decomposition of a difficult problem will be helpful in solving it, but some ways of pulling it into separate parts are much better than others. First and foremost, the parts have to *be* separate. If the division is not clean, one part will intrude on another, allowing the whole problem to sneak back into consideration. A second desirable quality of decompositions is

called "separation of concerns." It is very helpful if each part comes with a restricted set of important issues and characteristics. Then when thinking about (say) part A, it will be necessary to deal only with (say) issue X that is relevant to A, not issues Y and Z that apply only to other parts.

For example, imagine capturing a complicated story, perhaps a description of some real event that took place. The most obvious decomposition is temporal; as Lewis Carroll advised, one should tell the story as follows: "Start at the beginning, go to the end, and then stop." Perhaps the event to be described has some natural demarcations in time, such as a number of days over which it took place. But in a temporal description there is no separation of concerns: each part of the story is handled in the same way as any other part. Another decomposition of a story might divide it geographically, into parts that happened in different locales. Now there is separation of concerns, since some locations will be more important than others, and each location will have a set of unique features upon which to concentrate.

"While he was at his uncle's villa in Spain, Robert was never sure if his rusty Spanish let him grasp the situation."

Software engineering uses several dichotomies in decomposing the problem of developing software.

1.1.1 Management vs. Technical

Management and technical issues are completely different—the problems of running a large multiperson project are different from those of doing the work itself. Both are vitally important to software engineering. For an individual doing all software development, however, the managerial/technical distinction doesn't make much sense. One person could put on a "manager's hat" and tell herself what to do, then replace it with an "engineer's hat" to do what she was told and write a report about it, then switch hats again to read the report, and so on. But that's silly. "Management" for an individual simply becomes the activity of making a plan that can be consulted as the work proceeds.

Know How the Other Half Lives

This book is addressed to a technical audience, to engineers. One of the most important conditions of engineering employment is the freedom to *be* an engineer—to work with technical problems, and attack them in a skilled, technical way. The paradox is that this desirable situation must be established and maintained not by engineers themselves, but by their management. A good engineering manager in a company that values engineering spends a lot of time creating the right environment in which subordinates can work productively. The manager deals with the politics, the larger business and personnel issues, and the day-to-day nontechnical disasters that engineers don't want to face. The manager acquires resources like state-of-the-art workstations for engineers to use. The manager insulates engineers from the demands of other people in the company, who have their own important problems (like how to pay the bills next month, how to please a demanding customer, etc.). The manager is a champion for ideas coming from engineers, and sees that they get a hearing up the administrative chain.

Enlightened engineering management is not an altruistic or one-sided enterprise. Engineers need managers, and companies need engineers who are not managers. The reason is once again rooted in separation of concerns. A good engineer is not a good manager, and vice versa. Hard technical problems require concentration and all the attention and skill that a human being can bring to bear. Everything else must be pushed aside to solve them. So the best engineers have no time for anything but the problem on which they are working. However, everything else *cannot* be pushed aside—and if someone doesn't deal with the real world as it impinges on an enterprise, that enterprise will fail. The contract will be lost, the customers will be unhappy, the company will go bankrupt. However much engineers would like to deny it, even their own work needs directing and controlling. It is all too common for even the best technical people to get lost in some trivial or misdirected aspect of a problem, and waste their time while important other aspects are neglected. It is a manager's job to see that this does not happen.

One of the most important decisions an engineering student can make during schooling is a choice of "which side are you on?"—to be a part of the technical work, or a part of managing that work. A few people can do both; most of us can't. If your bent is to be an engineer, you'll be unhappy as a manager, and always trying to get your fingers into the "real work." If you lean to management, you'll be impatient with the narrow technical view, which seems to ignore all the political and human issues that "really matter."

This book is almost entirely concerned with the technical details of the engineer's craft. Some readers will have made up their minds which side they want to be on, whereas others will keep their options open. The authors are themselves solidly in the "engineering" camp. Learning about what software engineers do is one way to help make the choice. However, it is often important to "know how the other half lives." Managers are human, and not very many of them hold the ideal view that they exist to help their engineering subordinates. The majority of well-meaning managers probably believe that it is the other way around. A minority of managers are not well meaning, but more interested in their own power and their own careers than in any technical issues. In dealing with management, engineers can benefit from understanding the larger context in which they work. In that spirit we have included some information about the way in which software development fits into the world.

You Don't Have to Love 'Em

For all that it helps an engineer to understand the larger context in which engineers must work, and it is also crucial that managers understand the nature of the engineering that they supervise, there is a real and natural

antagonism between the two camps. It is very human to blame anyone but yourself when something goes wrong, particularly if the other person is not one of your "own kind." So engineers grouse about their incompetent managers who couldn't even write a "Hello World" program, and managers sometimes think of their employees as children who must be told to use the toilet.

Both sides could be better critics. An engineer might ignore all the company politics, yet can be relied upon to tackle and quickly solve hard technical problems that baffle everyone else. Such an employee may be personally repugnant or incomprehensible to a manager who is the exact opposite, but the manager would be a fool not to value the engineer and help with the necessary politics. What a manager would better criticize is an engineer who doesn't solve assigned problems, and the better politician that engineer is, the worse for everyone. Similar remarks apply to an engineer's criticism of a manager. Since the manager isn't doing the programming, such skills may not be relevant. But a manager who doesn't provide needed resources, or who abuses supervised employees by blaming them to save his own neck, or who does not competently convey employees' concerns to managers higher up in the company, is not doing his job.

A little understanding goes a long way in the difficult relationship between manager and engineer. It is difficult to be generous and forgiving if one does not understand what is driving the other side.

1.1.2 Teamwork vs. Individual Effort

It is entirely legitimate to see engineering as a separate discipline whose concerns are often diametrically opposed to those of management. An engineer who believes that a similar dichotomy exists between working in a team and working alone is on much shakier ground. The psychological component of the issue may be the same: engineering does attract the "loner" who is not skilled at working in a group, and group effort does detract from concentrating on the technical details of a problem. However, it is increasingly true that teamwork skills are valuable—even necessary—to the individual in developing software. The analogy "management is to engineering as teamwork is to individual work" breaks down for two interrelated reasons:

1. There is real technical substance to teamwork; unless engineers know how to do it, it can't be properly done.
2. Although the concerns of a team are somewhat different from those of an individual, they cannot be separated and assigned to someone else; teams are made up of individuals.

Thus engineers need to know about and practice teamwork.

However, this textbook contains little more about working in teams than it does about management issues. Part of the reason is that the skills required

are based on psychology more than on mathematical science. Though we feel comfortable including an introductory section on (say) first-order logic, we cannot include a psych primer. The more important reason for omitting technical material on teams is that the skills are best learned by participation in a team itself. So a textbook being read by a person sitting alone is just the wrong vehicle for teaching those skills. There is one exception to our lack of coverage: software inspection is so important that we give an introduction in Section 5.3. Inspection illustrates the technical nature of team activities: the psychological aspect is important, but equally important is the engineering skill of the participants. A team could be assembled to do software inspection, trained in the dynamics of the activity but without engineering skills. They could do a classical, textbook-perfect job that would nevertheless be almost worthless because their work would miss the point of the inspection. It is the engineering that is being inspected, and only engineers can do the job. (We are not denying the necessity of psychology: engineers can also fail miserably at inspection for lack of this knowledge.)

There is one valid part to the management/teamwork analogy. A person may not want to work in a team any more than he or she wants to be a manager. Such a bias would dictate looking for employment in smaller organizations where teamwork is not valued. But even here, the analogy breaks down because even in a startup company with a handful of technical employees, teamwork like inspections can be immensely valuable. Thus someone who likes the idea of being part of a team is really not restricted to larger companies or projects.

1.1.3 Different Tasks—Development Phases

Finally, we come to the separation idea that is at the heart of software engineering. It is as useful to an individual as it is to a development group: the development task is arbitrarily split into several distinct tasks. These tasks are called "phases of the software life cycle." For example, the "design" phase is concerned with dividing the programming task into independent parts, which is quite separate from doing the programming in the "coding" phase. Some of the boundaries between life cycle phases are pretty arbitrary or fuzzy, particularly when an individual is doing the work rather than a group of people. But there is a principle for marking off the phases: each results in a self-contained "document" that encapsulates part of the effort of developing software. This document concisely captures all the work done up to a point (which defines the end of the phase); after that point the document is consulted without rethinking any of the decisions that went into it. For example, part of the "design document" describes the modules (subroutines) of the software. All the work of choosing appropriate modules and defining their interconnections is called "design" and captured in a "design document." Once the design phase ends and the "coding" phase starts, the

module breakdown is accepted as gospel by the programmer. To question it would be to open up all the old issues and decisions again. Using the design document makes for efficient programming; the concerns have been separated and part of the problem solved.

Of course, it can happen that mistakes in one phase come to light only later on. It is in the nature of problem decomposition that ignoring all but one part blinds the developer to some information. When such mistakes are made, the decomposition may prove counterproductive. For example, if the interconnections of the module design mistakenly omit something, and this is only discovered near the end of the coding phase, then much work of both design and coding may have to be done over. In that case it would have been better not to separate the phases, but instead to deal with the whole design/coding problem at once. There is no guarantee that mistakes will not be made. But here is where the discipline of software engineering makes its contribution. The design/coding separation is not arbitrary, and engineers can learn to do a good job of design by studying its principles. Then in most cases mistakes will *not* be made, or at least lessons will be learned from mistakes so that they are not repeated, and the problem decomposition will usually work.

It is natural to describe software development by describing each phase in an ideal time-order of occurrence, and indeed that is the plan of this book. Real development seldom takes place in this ideal order, one phase neatly following another. But if it did, there is a name for the ideal process: "the waterfall model." It has been sensibly suggested that this nonexistent ideal is a great way to describe development, even if the development didn't work that way. Each phase has a distinct goal, and a distinct product that defines it, showing the separation of concerns involved.

However different the phases of development may be, they share many principles, important considerations that apply to software and its development in general. Rather than repeat these principles for each phase, we have collected them in Chapter 2. The principles express important ideas about making software, and in a sense the definition of a "life cycle" is an artificial means to create phases to which the principles apply.

1.2 Phases of the Software Life Cycle

The software engineering literature is inconsistent in naming the parts of software development. There is no real disagreement on what actions take place, but there is confusion about just how many distinct phases these actions should be broken into and what names to use for the phases. As usual in computer science, the same names have different meanings in different papers and books.

The first confusion arises in the name for the whole process. Some people call the complete construction of a software system "software development." Others refer to the process as software "design," "construction," or "building," or combinations of such words. Here we choose to use *development*. (And in particular, we do not call the whole process "design" because that will be our name for one part of development.) Together, the parts of development are also called the *software life cycle*.

A "phase" of software development is defined by the existence of a product (artifact, document, work product, etc.) that results from that phase. The goals and concerns of the phase are embodied in properties of that product. Once the phase is completed, its product serves as the basis of the subsequent phase(s). The whole point of dividing the task of development is to get intellectual control of the process by "separating concerns." That means part of the work is considered by itself, completed, and encoded in a document. Later, when other parts of the work are done, that document is consulted as a given—the earlier work is not reexamined or redone. For a small project, this rigid organization is a handicap—it is better to have all aspects of the work fluid and able to change as other aspects are considered. But for a large project, too much freedom is a curse: there are so many choices to be made, and so much interaction among the choices, that one must be arbitrary (and perhaps wrong) in making them and then must stick to early decisions, or the work will go round and round seeking the "best" solution, and never be finished.

There is another, less obvious advantage to be gained by separating development into distinct phases. People developing software *will* make mistakes. Working on a self-contained part of the process allows mistakes to be made (and discovered) as early as possible. Correcting a mistake is far easier at the beginning than at the end. Imagine that in the process of testing a software system, it is found to fail, and the failure is traced to a misunderstanding about what the software was supposed to do. Then the developer must go back to square one and do everything over, perhaps rewriting much of the code. If that same misunderstanding was discovered at the outset, there is almost no investment yet made, and it will be easy and cheap to correct.

The remainder of this section describes the processes that go into any software development, without giving them any definitive names, and then summarizes the life cycle that will be the basis of this book. Most of the difficulty about names comes at the beginning of the

1.2.1 Information Exchange Between Developer and End User

The end user is the person for whom the software is to be developed. A classroom instructor might be the user (for an assigned project); the developer might also be the user (for a personal utility program). In commercial software the user is a customer who may be someone in the same organization

as the developer (so-called in-house development); or people in a different organization that is contracting to have the software developed; or even an over-the-counter buyer (for so-called shrink-wrap software). In any case, the user needs to tell the developer what is wanted, and this is the first and probably most important phase of development. In the case of shrink-wrapped software, the real end user is not present in the discussion but only suggested by a marketing representative or by the developer's own personal idea of what will sell. Sometimes the product resulting from the user/developer dialogue is a binding contract signed by both parties—a contract to which the developer may later be held by law. However, it is more usual for the agreement between the parties to be less formal, only a nonbinding description of what is required. The common situation is one of good will on both sides: the user believes that the developer will try to do the right thing, and indeed this is the developer's intent. When things go wrong, it is more likely to be a mistake than a conspiracy.

Never attribute to malice what can be explained by incompetence.

Perhaps the most common name for the product that results from user/developer interaction is the "requirements document." One of its most important characteristics is that in its entirety it should be comprehensible to the user.

1.2.2 Complete Problem Description (Developer Only)

Some aspects of what software must do are not of concern to the user, but only to the developer, and may be expressed in terms the user does not wish to master. These requirements therefore do not enter into the user/developer dialogue, or when they do, the user dismisses them as "don't care" conditions. The developer, however, must settle all such details before proceeding. Thus there may be additional "requirements" placed on the software, and a separate phase to create what is often called a "specification." Or the product may be called a "design," with adjectives like "architectural" or "high-level." The distinction between this and the previous task is that here the user is absent. However, often when the developer begins to add these "don't care" requirements, it is obvious that the user *will* care, so the case is re-presented to the user, with some specific choices.

Hence a developer-only specification may put itself out of business: part of it might be added to the user requirements dialogue, whereas the rest might be moved to a subsequent phase. The test for what should be in a specification is:

> It is supposed to say exactly *what* the software is supposed to do, but to avoid as much as possible saying *how* that functionality will be attained.

The "how" belongs in subsequent tasks of development.

1.2.3 Break Down into Detailed Assignments for Programming

For a large, multiperson project, it is essential that the programming be done in parallel by people who need not talk to each other about the work; otherwise, the use of more people will result in less work getting done! Breaking the programming work into independent assignments is therefore the next task following functional specification. Even for one-person projects, however, this decomposition is important in keeping control of the work. If one is continually revising the pieces to take account of the other pieces and changes in them, nothing gets finished. In this breakdown phase, components are described, and each component has an interface to the others, which must also be described. The process is usually called "design," with adjectives like "low-level" or "detailed." The design process is concerned with the "how" of the software.

However, each component of a design must have a functional description, which brings the ideas of the previous phase (specification) into play at the level of components. The role of the user and developer are played, respectively, by the designer and the programmer. Because the designer is very knowledgeable about the programming to follow, the actual programmers who will implement the design may not be consulted, but only imagined by the designer.

1.2.4 Programming

From the component specifications and the design that interconnects the components, programming language code can be written. Insofar as the programmer assigned to this implementation has any questions or difficulties in deciding what is to be done, one of the earlier phases is at fault. Technically, when such a question arises, the development process should stop and the question should be settled (usually by consulting the end user). In practice, the ideal is realized only for "show stoppers"—problems so severe that no one can imagine just going forward. For less severe questions, the programmer must often make a decision and implement it. However, this decision *must* be communicated to the user. Putting a comment in the source code is entirely inadequate; at the very least the decision should get into the users' manual, and best of all, it should be described in a special notification to the user.

The product of the programming phase is, of course, a program executable on hardware. Often the source listing is taken as the product, but it is almost always assumed that high-level code has been compiled successfully. There is disagreement about whether the code should have been executed, but if the program is assumed to be tested, the testing is not extensive or necessarily well organized. Such testing as is done is rather like an extension of the syntax checking of compilation.

1.2.5 Testing

Once a component has been coded, it can be tried on examples in isolation. The components may be subroutines, data structures, header files, or combinations of pieces. The generic term *module* is used for complex components although this word also has a technical meaning as a particular kind of component called an "abstract data type." The term *unit* for an ill-defined component is often used, and component testing is almost universally called "unit testing."

When all components are ready, the composite system can be assembled and tried. If not all components are assembled, but only those that depend heavily on each other for some aspect of operation, the trials are sometimes called "subsystem testing." When the entire piece of software is tried, it is "system testing." Both of these aspects are covered by the term *integration testing* since units are integrated for the larger tests.

All parts of the testing phase include the computer-intensive aspect of executions. However, executions cannot be done without the input values for the components, the subsystems, or for the complete program. Furthermore, the executions are pointless unless the results are checked for correctness. These two tasks of generating test inputs and examining execution outputs usually fall to human beings, making use of the documents from earlier phases. Of course, results cannot be examined until the tests are actually performed, necessarily at the end of the development process. But finding test inputs can begin at the beginning of development. Each activity in development suggests tests, and if these are recorded along the way (in what is usually called a "test plan"), then testing can begin as soon as code is available.

The product of the testing phase is in one sense the tested software. But a better way to look at testing documentation is as an audit trail that establishes what was done. Thus the complete test plan (which has been growing throughout the previous parts of the life cycle) and a report of tests and their outcome are the testing "products." Reporting on tests means more than just listing them and giving the history of success and failure. The test report should seek to convince the reader that the tests conducted are meaningful, real evidence that hard work has been done and that the software was found to be of good quality.

1.2.6 Changing the Program

When a tested software system is delivered to its end user(s), it is more accurate to say that its life is beginning than that the life cycle is at an end. The user immediately discovers that what's wanted is something rather different from what's been delivered, or, serious deficiencies in performance or functionality are uncovered when the software is used for real work. Changes

are required, and all such changes are lumped under the somewhat inaccurate term *maintenance*. The analogy is to the "maintenance" of physical equipment (e.g., automobiles), which refers to actions compensating for or preventing wear and tear. Software does not wear out, and its "maintenance" is more like "redesign." But the word *maintenance* is solidly established. It has been suggested that maintenance history describes how useful a piece of software is. If it has not been repeatedly changed, it simply is not being used.

Just as problems that arise in coding should ideally send development back to the first requirements/specification phase, so should most maintenance begin with user dialogue and proceed through an abbreviated version of the life cycle. In reality, maintenance is often viewed as a low-cost activity in which working code is "patched" to alter its properties, and specifications and designs are neither examined nor updated. Software that is maintained in this way soon becomes so fragile that it is impossible to alter because the accumulated undocumented changes interact with each other, and cause any new change to fail in some obscure, unsuspected way. When a system reaches this state, it is literally "unmaintainable" and *must* be redone from scratch. The economy of cheap maintenance is therefore thrown into doubt. In practice, unmaintainable software, if it can still be used and sold, is frozen, and an attempt is made to document all its existing deficiencies.

All existing bugs are hereby declared features. You may use them without fear that they will ever be corrected.

In a textbook for the individual, the "maintenance" phase is treated only briefly although it is of the utmost importance in real-world projects. In some organizations, software development is nothing more than maintenance because there is so much existing software that the resources for creating new programs are all taken up dealing with the old ones. Maintenance is less relevant for students in the classroom. Even in "project" courses like compiler writing, students do not typically keep or alter their own software—they turn in a completed assignment, and that's the end of it. In an industrial software development setting, this is usually referred to as "throwing the software over the wall"—the developer contracts to deliver the product, but not to maintain it. A cynic might say that by doing a poor job in the first place, the developer can hope for a lucrative follow-on contract for maintenance since no one else will be able to understand or fix the delivered software!

Maintenance has a hidden role throughout the life cycle. Since all useful software ends up being "maintained," many of the goals in all phases of development are really maintenance goals. Software should be constructed so that it can be changed easily. Furthermore, whenever mistakes are made within development, and it is necessary to return to an earlier development phase to correct them, the work being done is just like "maintenance." Many of the best ideas in software engineering have a dual role: (1) they help the developer stay in control of the project, and (2) they make it easier to

return to the project later for maintenance. Thus maintenance will come up implicitly in the other phases.

1.2.7 Naming the Development Phases

In many cases of individual software development, the dialogue between user and developer is abbreviated. For example, a course instructor (the user) makes assignments (requirements) that students (the developers) are supposed to accept as perfect (although they seldom come close to perfection). Thus all the "what" about the software, its required functionality, is given by the assignment; in this case, the "requirements" are more like a "specification." For personally developed software, the "user" and "developer" are the same person, making the "requirements"/"specification" distinction unimportant.

When there is reason to keep the user-oriented problem description separate from the one the developer works with, we call the two activities *requirements* and *specification*, respectively, and talk about two separate documents. Often there is no distinction, but we still need to talk about "what the software is supposed to do," and for that we use either *requirements* or *specification* interchangeably, or sometimes even *requirements/specification*.

With or without user dialogue, the skills of constructing good requirements and specifications (and of criticizing poor ones!) are of great importance to individuals. To successfully develop software, it is essential to know what that software is required to do, and despite the best intentions, any requirements or specification document is almost certain to be deficient. Course instructors think hard about their assignments, but that is insufficient; to paraphrase the saying about representing yourself in court, "the developer who writes code for himself has a fool for a user." The sooner one can discover and correct deficiencies in requirements and specifications, the less work will be wasted.

Making a *test plan* is not a self-contained phase of development, but instead it goes on in parallel with the other phases, beginning as soon as work starts on the requirements/specification. Looking for tests that will exercise the required functions, as well as thinking about what outcome those tests should have, is one of the best ways to discover the deficiencies of a requirements/specification.

Our name for the "how" phase of software development, in which components and their interfaces are defined (and specifications given for each component), is *design*. Design includes more work on the test plan (to which component tests can now be added). The names *coding* for the programming itself (with more additions to the test plan), and *testing* where the test plan is used, are nearly universal. Finally, the tested program is delivered

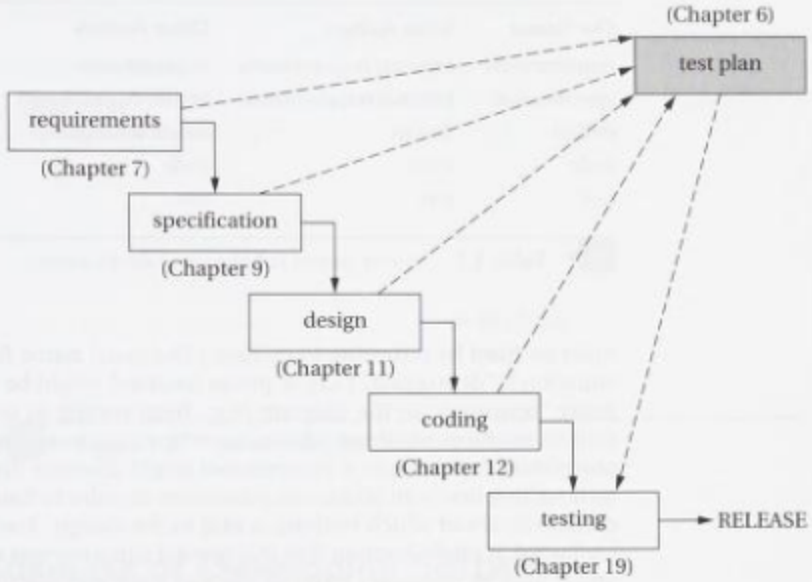


Figure 1.1 A waterfall diagram of software development.

and development is complete so long as everyone pretends that there will be no maintenance.

In talking about phases of software development, the name of the phase, say, "design," is technically an adjective: "design phase." But the name is commonly used alone, as a noun. Everyone says things like: "Design is taking longer than we expected." Unfortunately, the document that results has the same adjective: "design document," and the same abbreviated way of speaking is common: "The design didn't get updated to reflect those changes." So when you see "design" out of context, it might mean the phase, or it might mean the document, or (most likely) the person speaking or writing isn't paying attention, and it doesn't really mean much of anything.

Each phase of development is the subject of chapters to follow, as indicated in Figure 1.1, which shows the phases of development in what has come to be called a "waterfall" diagram. The name comes from imagining that work accumulates in each phase, to "spill over" into the next, in the form of a document capturing the earlier phase. The waterfall diagram fails to show the part of development where mistakes are made and corrected. This happens most frequently inside one phase, and often between adjacent phases. For example, testing uncovers a problem in the code, which

Our Names	Some Authors	Other Authors	Yet Other Authors
requirements	external requirements	requirements	specification
specification	internal requirements	architectural design	high-level design
design	design	detailed design	low-level design
code	code	code	implementation
test	test	test	test

Table 1.1 Some names for phases of development.

must be fixed by returning to coding. (The usual name for that particular situation is "debugging.") Cross-phase feedback might be shown by arrows going "upstream" in the diagram (e.g., from testing to coding). The most serious mistakes are those whose correction requires retreating more than one phase. For example, a programmer might discover that the subroutine he is writing needs an additional parameter in order to handle some special condition, about which nothing is said in the design. Tracing the problem backward, it might happen that this special situation was never thought of, and it may be necessary to go all the way back and ask the end user what should be done. On the diagram, this could be shown as a feedback arrow from coding to requirements.

The worst mistakes that can be imagined in software development come to light only after the software has been delivered. In the classroom situation, they are the problems that flunk the student whose program doesn't work, problems of which the student was blissfully ignorant. "Maintenance" is the misnomer for changes made after delivery, but these could be captured as feedback from the "afterworld" of the released software. For example, "corrective maintenance," a fancy name for fixing a bug that wasn't caught during development, is feedback from release to coding. "Adaptive maintenance," a change to some feature of the software as delivered, is feedback from release to requirements/specification. There are potential feedback arrows connecting all the phases in the waterfall diagram, but they are not shown in Figure 1.1.

Perhaps the lack of standardization is the result of using shallow, poorly understood, and imprecise ideas?

It is a measure of the youth of the software engineering discipline that the names for parts of development are not standardized. In this text, we will try to use the names from Figure 1.1 consistently, but the reader should be aware that in reading other books and papers a wide variety of names will be encountered. Table 1.1 gives a few of the possibilities, and the names in the table occur in combinations not shown as well.

There would be less disagreement about the names of the parts of software development if fewer distinctions were made. Most people could probably agree on a waterfall diagram that looked more like Figure 1.2. The labels beneath the boxes refer to the parts of this book.

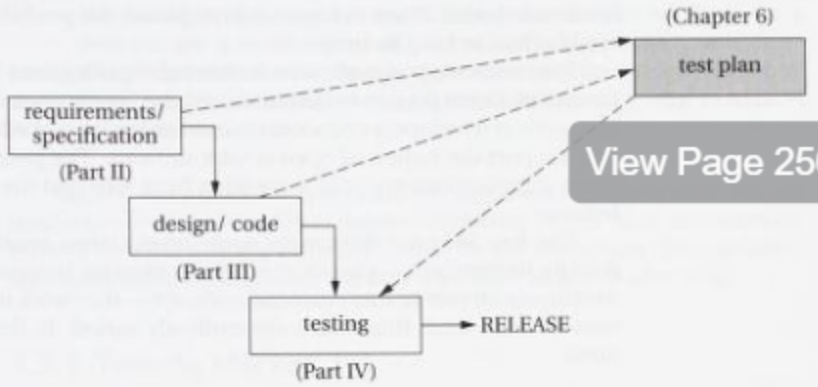


Figure 1.2 An abbreviated waterfall diagram.

1.3 The Business of Developing Software

Throughout this book, the development of software is treated as a technical problem, and the skills necessary to solve that technical problem are the book's subject. For what is so aptly called "an enthusiast," someone who loves to do the thing for its own sake, there is no more to software engineering than technical skill.

Computing has always had its share of enthusiasts—many people are fascinated by computers and what they can do. There is today an informal banding together of computer enthusiasts in what is called the "free software movement." "Free software" means that the end user of the software pays nothing for it, and anyone is free to modify the existing software and contribute to its development. The source is readily available to all users and developers, so there are a large number of people looking at the code. Although businesses can muster only small teams to develop and test their code, the entire world of enthusiasts and developers is available to write and test free software. Defects can be quickly and effectively resolved and distributed. (The other side of the coin is that chaos can ensue unless there is some control on changes made.)

Some businesses have embraced the idea of free software, but how is it possible for anyone to make money on free software? There are two ways: (1) A commercial organization can develop and sell software that runs "on top of" free software. The many enthusiasts using the free software are potential customers. (2) The pool of enthusiasts is a resource for a company that wants to contract for software development, for example, to support a specialized

hardware device. There is far more expertise in the pool than any company could afford to keep in-house.

Free software proponents have formed organizations to advocate their ideas. The Open Source Foundation and the Free Software Foundation are consortia of developers and sometimes even businesses who fund, advance, and support the notion of open source software. The people who work for these organizations are paid something from fees that the members pay to belong.

The free software movement sometimes claims another kind of freedom for its members—a freedom more like anarchy. However, most software developers answer to some external authority—they work in a software business of some kind. Businesses are endlessly varied. To list a few common ones:

Garage. The developer works informally and hopes to sell a product developed without any formal business structure. The extreme case is an enthusiast who plans to freely distribute the software, with no more than fame (or at least notoriety) as the reward.

Software startup. A garage business can grow into a commercial enterprise that employs a few people and sells a small range of software products. However, there is little formality in the working arrangements, and all the principals in the company share all the business roles.

Microsoft. Enough said.

Contractor. Contract programming occurs in many ways, but the archetype is an independent consulting firm, whose employees are assigned to a software development project that the firm delivers under contract. Here there are two businesses clearly separated, the contractor and the customer. The contract situation is unique in that maintenance of the delivered product may very well be ignored or contracted out to a different company.

In-house. In the early days of data processing using very expensive mainframe computers without much applications software in place, any computer system had to be written by those who wanted to use it. Data Processing Departments—now more often called management information systems (MIS) departments or information technology (IT) departments—were established to write and maintain the applications software the company needed. These in-house organizations are still around, more and more devoted only to maintaining the existing software. But organizations still have a need for tailored software, and any large company has its development staff, under whatever name.

Embedded-software developer. Embedded software (and the computer hardware may be "embedded," too) is part of a larger product. For example, the engine control computer and its software on a new car are a relatively small part of the car manufacturer's overall effort. Embedded software must interface smoothly with other, more conventional engineering systems designed by (say) mechanical engineers.

Each of these business types, and the myriad of variations on them, are concerned to a greater or lesser degree with issues "larger" than the technical ones of constructing the software. Except for the enthusiast, if these issues are neglected or mishandled, the business won't be around for long.

1.3.1 Time to Market

Software originates in the idea that a problem could be solved using a computer, and culminates in a working program that solves that problem. In a business, the program is in some way "delivered" to a customer. "Time to market" is a name for the interval between idea and delivery. For consumer software, it is the time that elapses before the product can be sold. The need to minimize time to market is a potent force controlling software development. It is said that customers always want delivery "yesterday," and they are not at all interested in the developer's difficulties in meeting that date.

It should be obvious that an artificially imposed development time limit may be impossible to meet, or if it is met, the delivered software might work badly, if at all. There is an engineering saw that in any product development it is possible to have any two of: schedule, budget, quality. (In complex enterprises like software development, however, increasing budget to meet a schedule but retain quality may not be an option. Fred Brooks is the author of a famous essay, "The Mythical Man-Month," that argues that adding personnel to a late project only makes it later. So perhaps one should say for software engineering it is possible to have one of: schedule, quality.) The only thing that can be done to hurry along a schedule without making a mess of the product is to shed features or performance. If overambitious plans are scaled back, it may be possible to deliver a good-quality product—just not the one originally planned—on time.

Time to market figures in even the least businesslike software development. An enthusiast working only for herself might rush things in order to get to the exciting part where the program is tried out. The compromises she accepts along the way may not matter—perhaps they could be revisited and mistakes corrected. But software bugs have a way of hiding where they cannot be found, and the rework could be so difficult that her project, once brought to premature completion and failure, is never finished.

A student working on a programming assignment has a fixed development time that ends on the assignment due date. The instructor who makes the assignment may or may not have allowed enough time; students have

many things to do, and may not be able or willing to use the full period. (It is a rare student who then decides to begin at once and finish early.)

In a business, someone is responsible for selling the product, and the "marketers" are advocates for the shortest possible time to market. Engineers are usually opposed, asking for more time to do a better job. The two sides try to negotiate a compromise that will produce a good enough product in the shortest time. The conflict is a real one, but within a company everyone ought to be on the same side of trying to make the company prosper. It is one of a manager's toughest jobs to facilitate the discussion between engineering and marketing, and ultimately to be responsible for the decision that both will have to live with. Give the engineers too much and the excellent product they produce will not sell because competitors get there first; give marketing too much and the product will be so shoddy that it won't sell. Honesty on both sides is essential; all too easily the discussion can become a power struggle or a clash of ideologies. Engineering must be honest about the time required, and often the hardest part is asking for enough time—engineers are notable optimists, and always hopeful that they can perform miracles. Marketing must also be honest about how long the market will wait, and must not discount engineering concerns about quality by thinking: "We can sell anything if we just get it soon enough."

1.3.2 Perceived Software Quality

Like beauty, the quality of an engineered object is in the eye of its beholder (user). There are many dimensions to software, many characteristics that may or may not be important to different people. These characteristics are sometimes referred to as the "-lity"s because their names have that ending. Some of them are:

Functionality. What the software can do, its "features." In view of the all too prevalent possibility that software may fail to work, perhaps one should talk about its *advertised* features.

Reliability. The probability that software will not fail over a period of time. Reliability can be expressed as a "mean time to failure," how long on average it can be expected to work perfectly.

Dependability. The likelihood that the software will do what the user needs it to do. Dependability includes a weighting for the importance (to the user) of the task. Dependable software may fail, but not in a catastrophic way.

Usability. How easy the software is to use.

Interoperability. How well the software can be used with other software. Good interoperability means that input can easily be taken from other programs, and output can easily be sent to other programs. For example, a spreadsheet program should be able to send its output to a word processor program for inclusion in a report; a data collection program should be able to send data to a plotting program.

Maintainability. How easy the program is to change, both for fixing bugs and for adding to its functionality.

It is notable that "performance" of the software, in the sense of how quickly and efficiently it runs, is usually omitted from the list.

"Software quality" is elusive because it has so many dimensions. A user who needs interoperability may view a dependable program as poor quality when it can't be used with another as needed. Someone who finds all the functionality he needs will rate highly a program that another person thinks of as being of poor quality because it is difficult to maintain.

Software that excels in one quality dimension is likely to fail in another, just because every aspect of quality is different, and each takes time and trouble to attain. Time to market is the natural enemy of quality, and under time pressure, the development engineers may have to choose what can be accomplished and neglect something else. Furthermore, quality aspects interact in a pernicious way. To gain functionality increases complexity, which in turn tends to reduce maintainability and dependability.

Probably the most important dichotomy is dependability versus functionality. Some people don't want their software to let them down when they need it, and they don't care if it's a bit limited in what it can do; others are willing to put up with unstable software so long as it tries to handle everything, including the kitchen sink. It is reported that Microsoft's philosophy favors functionality over dependability; the overwhelming majority of service calls they receive are requesting not fixes for failures in their released software, but rather asking for new features. However, the advertising campaign for Microsoft Windows 2000 stresses its dependability, so the corporate vision (or at least the desired image) may be changing. A very similar tension occurs in the automobile market, and history there has shown that dependability can be sold, but only if features are not missing.

In every contest between an ugly, old-fashioned, dependable car, and a flashy one with poor dependability, the flashy one won.

Sometimes software applications dictate what kind of quality is needed. Microsoft seems to be right about its customers wanting functionality at the expense of dependability. Safety-critical applications like software embedded in medical equipment, on the other hand, demand dependability.

1.3.3 The Software Development Team

Software development is just one of the many human activities that must be organized to succeed—and it draws on the lessons of older cooperative enterprises. The two fundamental organizational paradigms are the "hierarchical" and the "distributed." In the former, people are placed in an organization chart that makes clear who is in charge of whom, and directives

flow down the chart from the top in ever increasing detail. In the less common distributed organization, a group of people agrees to work together by consensus and cooperation, but each freely agrees to his or her assignment, and there is no reporting chain or supervisory relationship.

The essential psychological fact about human beings working together is that an effective working group cannot be too large. When more than a handful of people must cooperate, the overhead of mutual communication becomes so high that little else gets done. So if a project requires dozens of people, it will have to be organized hierarchically, with each small group reporting to a supervisor, a few supervisors reporting to a manager, and so on up to the project manager. IBM mainframe software was developed by such groups with hundreds of people; in its heyday, DEC software for minicomputers was developed by much smaller groups, almost in a distributed organization, reporting to a single layer of management. Microsoft uses an organization in which each application product (like the Excel spreadsheet) has its own team, probably run in a distributed fashion for small teams, but hierarchically for large ones.

The early phases of software development demand a few highly qualified people to handle requirements, specification, and design. Coding and testing can be spread over a larger staff. Design organizes the programming effort into independent units for assignment to programmers who work independently; testing is an inherently parallel activity that requires very little organizing effort to utilize hordes of testers.

1.3.4 Managing Development

The first decision that a software development manager (or the company for which the manager works) must make is the extent to which a formally defined and controlled process will be used in development. If there is to be control at this abstract level, the company has to spend considerable effort defining exactly what procedures will be followed, in training engineers to follow them, and in training managers to track the engineers' efforts. Passing from an organization in which there is no well-defined process used for development, to one in which all the steps are laid out and monitored, is a very large order. The Capability Maturity Model (CMM) described in Section 4.1.3 is a rough guide to five levels of organization that a software development operation might attain, and to pass from one level to the next is estimated to take on the order of two years. Most organizations are presently at or near the first level; many aspire to higher levels, but few to the highest ones—it is just too far to go.

Suppose then that something like the waterfall model, as an outline of the process to be followed, is in place in a company. What does management have to do as development proceeds?

Allocate resources. Ideally, personnel can be assigned to development phases according to which ones can be done in parallel (test plan with design and coding, coding of independent modules), and staff can come and go on the project as they are needed. In practice, a fixed group of people is sometimes assigned for the duration of the project, and some of them may be under-utilized or overworked as it proceeds. Continuity in the group is important because there will be mistakes and feedback loops among the phases in which detailed knowledge, such as only an original team member possesses, will be essential.

Computing and staff support must also be allocated to the project, and there may be substantial lead times in obtaining them. For example, if a project is to use automated testing (Section 21.3), support software will have to be acquired and installed, and engineers must be trained to use it effectively.

Monitor progress. A project manager lays out a detailed time-line for the work to be done, showing each activity and resource use as it is supposed to occur. Along the time-line, a manager wants to place as many "milestones" as possible—tangible items that the manager can use to monitor progress. The documents that end each phase of the waterfall life cycle are such milestones, and they are supplemented with interim reports.

Sometimes milestones are useful to engineers as well as managers although usually in a different way because their concerns are different. A completed test plan, for example, tells a manager that the requirements are good enough that later testing of the code can be carried out, and that the time-consuming testing phase has been prepared for in advance. Engineers are more interested in recording all the testing insights that can be gained from requirements before these are forgotten. But it is easy for management and engineering to be at loggerheads over a milestone—management insisting on it while engineering views it as unnecessary busywork. Both sides are right, according to their own concerns. Management needs to know how things are going. Engineering knows perhaps too well how things are going, and writing a report about it will take time that should be spent making progress. Engineering is at the mercy of management here, and the competence of a manager might be judged by how well he or she can evaluate progress without insisting on otherwise useless reports.

Assure quality. Software development not only has to be completed (preferably ahead of schedule!), but in the end the software must work well enough to be useful. If the developer has carried out a sensible testing phase, this ultimate measure of quality is probably known, unfortunately too late to be of much use. (And of course, if the testing

itself is of poor quality or has been eliminated because of a schedule crunch, nothing will be known until angry customers begin to be heard. The engineers, of course, will have a pretty good idea that all is not well, however little their managers know.)

What managers need as milestones throughout development are accurate indicators of how well the software to be finally delivered will perform. For example, what properties of a test plan, a milestone early in development, would point to a good final product or a poor one? There *are* ways to judge the quality of test plans and other development milestones, such as a design breakdown of code into modules. This book is devoted to explaining precisely what is known about good and bad software engineering. But unfortunately, the connection between doing good work along the way and getting a good result is a tenuous one. It is quite possible for engineers to go through all the right motions, their managers checking diligently all the way, yet still produce an unacceptable end product. For example, this could happen if a serious misunderstanding of the requirements occurs, or if a crucial design decision is wrong, or if code fails in some catastrophic way that testing did not detect. Everything in a modern software development process is designed to catch such mistakes, but no one believes that our best practices are good enough.

Furthermore, managers must rely almost exclusively on the good will of their technical engineering staff for any evaluation at all. Software development milestones like the test plan are complex and detailed, and although the people who made them usually know whether they are any good, it would be an unusual manager who would notice if his subordinates tried to cheat. Thus the force balancing the manager's ability to insist on a stupid report is the engineer's ability to fake the report without getting caught. An organization in which such games are played is in deep trouble. Under reasonable circumstances, engineers acknowledge the need for managers to know, and help all they can. Insofar as the whole development process works, and its documents and procedures are really useful to the engineers, the managerial overhead is small.

These management tasks are interrelated. If progress is not keeping up to the required schedule, more resources may have to be added to the project or shifted from one phase to another; or some features may have to be omitted from the software, or less time spent checking it.

The phases of the waterfall model are very useful to a manager because they provide self-contained checkpoints along the way to delivery. As each phase is completed, and with it the document describing that phase, a manager can examine these realizations in text of what has been done, and get a good picture of how well things are going. There is even a clever way to get expert help in assessing the test plan, requirements, specification,

design, and code as they are produced, the "software inspection" described in Section 5.3. Without the waterfall phases, a manager is reduced to asking engineers to estimate how far along they are. The folklore is that whenever an engineer is asked this question, the preferred answer, independent of how things are really going, is, "I'm 95% done." A manager has no way to check such statements. Months after delivery was supposed to occur, it's still "95% done."

How Does It Fit?

(These questions at the end of each chapter explore ideas presented in each section. They do not necessarily emphasize the most important ideas of the section. A more pretentious name would be "thought questions," to help you see if you understand the presentation. Solutions and hints for some of the questions appear at the end of the book.)

From section 1.1

1. "Flashbacks" are a device often used to tell stories. Discuss the use of flashbacks from the standpoint of separation of concerns. How would you paraphrase Lewis Carroll's advice if the story is to be told mostly in flashbacks?
2. When people form into antagonistic groups, they find it very helpful to call each other derogatory names. Sometimes engineers are referred to as "nerds" in this way. And if a manager is pretentious enough, he might be called a "suit." If you have made up your mind which side you are or want to be on, write a paragraph on "What I hate most about suits (or nerds)." (If you haven't made up your mind, flip a coin to choose sides.) Then write a second paragraph in which you explain why it is right, proper, and good that suits (or nerds) are as you have portrayed them in your first paragraph. Which viewpoint is most true?

From section 1.2

3. The *products* of the development phases are not shown in Figure 1.1. Copy the figure and add them to it.
4. The waterfall model of the life cycle (Figure 1.1) suggests a river and a series of dams. The usual argument for dams is that they provide hydroelectric power, allow navigation, and control flooding. Opponents of dams say they interfere with the movements of fish, destroy the beauty of fast-flowing rivers and drown vast areas of the landscape, and when a large flood comes, they actually increase rather than limit the damage. Write a short essay in which you try to give analogies in waterfall software development to each positive and negative aspect of a series of dams on a river.

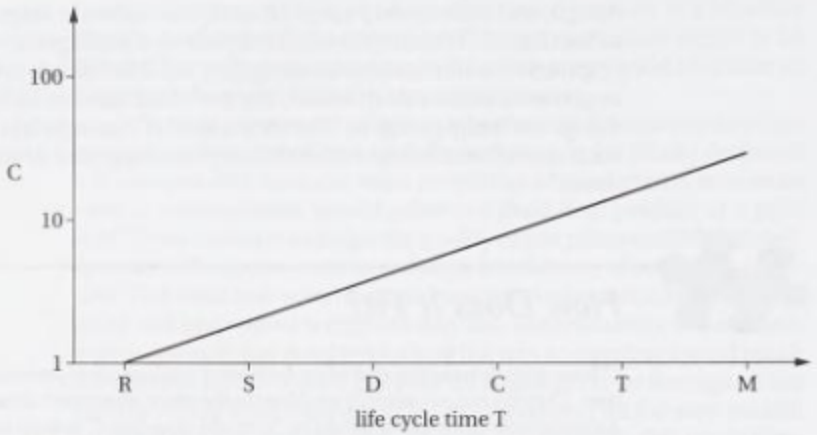


Figure 1.3 Relative cost of fixing a problem discovered at time T in the life cycle.

5. The curve in Figure 1.3 (after B. Boehm) shows the approximate relative cost C of fixing a problem discovered at time T in the development cycle. The rough positions of the phases are shown by their initial letters on the abscissa (e.g., "R" for Requirements). Notice that the relative cost scale is logarithmic. Thus, for example, if a problem found in the requirements phase cost $1000 to fix, it will instead cost about $10,000 to fix if not discovered until the coding phase.
   (a) Give an example of a requirements problem and explain the cost differential between discovery in requirements or discovery in coding.
   (b) Give a general explanation for why the graph has the shape it does (that is, why it is a straight line on semilog graph paper).
6. Suppose that someone invents a new phase of development called "bridge" that is supposed to go between "design" and "code." You are assigned to give a presentation about "bridge," but unfortunately you have only learned of this indirectly, and don't yet know just what is supposed to be done in "bridge." Nevertheless, you want to start preparing your talk.
   (a) In planning to be *critical* of "bridge," what kinds of arguments, based on general properties of the life cycle and the place of phases within it, might you use?
   (b) In planning to *support* the introduction of "bridge," what kinds of arguments would you use?
7. In Section 1.2.6, there is a cute saying about bugs and features. Give an example of a "bug" that a user might actually want to call a "feature," and explain why that is so.

8. Give a personal example of classroom "maintenance" of a piece of software, where you modified the results of one assignment to satisfy another, or changed an assignment's program to help with other work. Or, if you haven't done this, make up a plausible case where a student might do it.
9. Suppose that making the test plan *had* to be included as a separate phase in the waterfall model (Figure 1.1), connected as the other boxes are, not off to the side.
   (a) Where would be the best place to put it, and why?
   (b) Explain why making test planning a separate phase in the waterfall model is *not* a good idea.
10. A manager observes that the software engineers who work for him usually do not create test plans early enough in development. So he decides to force the creation of these plans by adding three new phases to the life cycle: (TP1) Test plan 1 is added between requirements and specification, (TP2) Test plan 2 between specification and design, and (TP3) Test plan 3 between design and coding. The inputs to these phases are the documents of the previous phase, and the outputs are three parts of a test plan. Explain what is wrong with the manager's idea.

From section 1.3

11. Does time to market figure as a force in contract programming? Explain how it does, or why it does not.
12. In the Challenger space-shuttle disaster, how did time to market enter, and who were the people analogous to "marketing" personnel?
13. Are power windows, seat adjustments, and outside mirror positioning in a car as dependable and maintainable as if these functions were done manually? Make an analogy to software.
14. Explain the difference between "reliability" and "dependability" by giving examples of a software system that would be considered dependable, but is not reliable, and vice versa.
15. Give some extreme examples of software that emphasizes one quality "-lity" at the expense of others. For example, how could it happen that a program is very easy to maintain, yet has poor functionality?
16. It is not easy to explain why performance is usually omitted from the "-lity" list. Other than its unfortunate spelling, can you think of reasons why?
17. Large hierarchically organized businesses have many "dead-end," low-skilled positions that small, decentrally organized ones do not. Comment on whether this has to be so.
18. Suppose that in reporting progress in the coding phase, an engineer says she is "95% done." Later on, when the code has been tested and released, a count of the lines of source code reveals that on her part of the work, 5200 lines were delivered, while at the time of the earlier report, she actually had 5300 lines. Was she telling her manager the truth?

19. When an engineer misspells "milestone" as "millstone," what might he be unconsciously thinking of?
20. Can you think of any instance in which an engineer might want to prepare a report that her manager did not request? Should she tell her manager about it?

Review Questions

(These review questions at the end of chapters are supposed to help with self-study. Their answers provide a rough outline of the chapter, and all major topics in the chapter are supposed to be covered by the questions.)

1. Name the phases of the waterfall life cycle to be used in the text, and briefly describe each in your own words.
2. The subsections of Section 1.2 do not give names to parts of the life cycle they describe. But names are given in Section 1.2.7. Make a list of which subsection(s) go with each name. A good way to display the answer would be to write the subsection numbers on Figure 1.1.
3. Contrast the manager's view of a software development project with an engineer's view.

Further Reading

(The list of readings at the end of each chapter includes two quite different kinds of citations, with brief descriptions of their content. One kind provides additional information, for example, in a standard textbook. The other kind of reference will take the reader far afield, and provide information about topics quite unlike software engineering. We note without comment that the latter references are usually more interesting than the former.)

The so-called software crisis in which the complexity of problems overwhelms software developers and makes their work late, over budget, and even inoperable is well described in a *Scientific American* article [Gib94].

Barry Boehm is often credited with inventing the name "software engineering." His seminal article on the subject [Boe76] makes very up-to-date reading, especially interesting in its predictions from 20 years ago.

Most textbooks on software engineering differ from this one in being concerned with the whole development process that involves an entire organization. This wider scope makes them thick books in which technical topics are not treated in depth. Perhaps the most used text is by Ian Sommerville [Som96].

A valuable collection of current readings is contained in a tutorial volume published by the IEEE (Institute of Electrical and Electronic Engineers, a professional association with a member Computer Society) [TD97].

Although he no longer considers himself a software engineer, Fred Brooks has excellent credentials: he was the manager of what may be the most complex computer system project ever undertaken, the IBM System/360. He writes with wit and common sense about the experience in *The Mythical Man-Month* [Bro75]. His more recent thoughts on software engineering (and the "software crisis") appear in a journal article [Bro87]. (And the two are available combined in an "anniversary" edition of the book.)

The Internet, and the World Wide Web that uses it, are becoming important sources of technical information, often rivaling libraries and journals. We have avoided making many references to Web sites in this text because the sites often change their name or disappear. But it is certainly appropriate to suggest that information about the free software movement be obtained from the Web, which is an important reason for free software's existence. So try the link `www.opensource.org` to learn more.

For learning about "what's *really* important" in software development, there are two excellent books by Weinberg [Wei99] and DeMarco and Lister [DL99].

References

[Boe76] Barry W. Boehm. Software engineering. *IEEE Transactions on Computers*, pages 1226–1241, 1976.

[Bro75] Fredrick P. Brooks. *The Mythical Man-Month*. Addison-Wesley, 1975.

[Bro87] Fredrick P. Brooks. No silver bullet: essence and accidents of software engineering. *IEEE Computer*, pages 10–19, 1987.

[DL99] Tom DeMarco and Timothy Lister. *Peopleware—Productive Projects and Teams*. Dorset House, 1999.

[Gib94] W. Wayt Gibbs. Software's chronic crisis. *Scientific American*, pages 86–95, Sept 1994.

[Som96] Ian Sommerville. *Software Engineering, 5th Ed.* Addison-Wesley, 1996.

[TD97] Richard H. Thayer and Merlin Dorfman, editors. *Software Engineering*. IEEE Computer Society Press, 1997.

[Wei99] Gerald Weinberg. *The Psychology of Computer Programming, silver anniversary ed.* Dorset House, 1999.

chapter 2

Some Principles of Software Development

In the chapters to follow, we will consider each of the life cycle phases in turn, giving the rationale for its separate existence, and describing how to carry it out in practice. In this section, we present a number of general principles that will appear again and again as software development is described. These principles are behind the phases of the life cycle itself, and they arise in the mechanics of every phase.

2.1 Intellectual Control

Only complex software needs to be engineered. In simple cases, "craft" techniques are better. The analogy to mechanical engineering aptly describes the situation:

> When making a relatively simple tool or gadget, a craftsperson works by "fit and try" methods, shaping and adjusting the parts until they work properly together. Craft objects are often pieces of art when the craftsperson has talent for beauty as well as function. But for more complicated objects, craftsmanship fails. When there are too many parts, and too many interdependencies between parts, there must be an engineering plan to which a mechanic works. With luck and experience, fit-and-try methods can succeed, but they are wildly inefficient compared to engineering, because when the final parts do not fit, the whole must be done over. However, attempting to apply engineering techniques to simple objects is also inefficient, because the extra work is wasted; the object can be built in the time it would take to plan its building.

It is the same for software. A simple problem can and should be solved directly by programming—its "life cycle" consists of just coding and testing.

with feedback between them when something doesn't "fit." Software engineering is for problems in which the craft method leads to extensive programming that must be done over and over because it never quite reaches its goal.

Students are usually taught to program in the "craft" style. They begin with simple problems and convert them directly into simple programs. Perhaps this is a good way to learn programming, but it establishes a false confidence that any problem can be solved directly. When a student has successfully completed a course in data structure programming, and has constructed programs of perhaps 2000 lines of code for somewhat contrived class assignments, he or she is ill prepared to tackle the development of small- to medium-scale software. Writing a compiler or an operating system kernel, which is expected of computer science undergraduates, *can* be done by many students without using software engineering. But some students find it impossible to "get it to work." We believe that some measure of software engineering will help, and that even those who don't need the engineering techniques will find them to be timesavers. As the problem grows, engineering becomes more and more valuable, until for "large" systems it is indispensable. It is simply impossible to build a commercial airplane without mechanical engineering; it is equally impossible to write the flight control programs for that plane without software engineering.

2.1.1 Complexity and Control

Problem complexity is the root of the problem of software development. When a human being attempts to solve a complex problem directly, failure comes from a loss of *intellectual control*. The person cannot see far enough ahead to know if decisions being made now will prove correct in the end, and thus may spend a long time and extensive effort pursuing a nonsolution. By the time the approach is seen to be inadequate, too much time and too many details have passed to be able to see exactly what went wrong, so the next try may be no more successful. In short, the person cannot fully grasp either problem or complete solution, and so gets it done only by good luck. Those who have worked in situations where they do not have intellectual control—where they *know* that their best work may be wasted—know how uncomfortable it is.

To gain intellectual control of a complex problem requires that the complexity be reduced. There is a respected minority of computing professionals who believe that this should be done by refusing to accept problems for software solution that are "beyond the state of the art." They are a minority because the forces that drive software development encourage grandiose projects. Software has handled "impossible" problems in the past, sometimes with apparently magical results; it is our strongest, most flexible tool for doing difficult things; the payoff can be immense, so why not try? Those who believe that some problems are just too hard may be right. Some optimists have gone bankrupt trusting to software solutions that did not materialize on schedule. Some projects have spent a great deal of money with nothing to show for it. Nevertheless, the majority position is that software professionals do not choose the problems on which they work, and hence complexity must be handled in some other way.

A general in the Reagan administration "star wars" program is reported to have said that if scientists believed the project impossible, he could certainly find others who would take his money.

Intellectual control must be gained in spite of problems in which it seems unattainable. The problem decomposition of the software life cycle is itself an attempt to reduce complexity and gain intellectual control. And within each phase, other decomposition techniques are used with the same intent.

2.1.2 Language Betrays Us

In a famous interchange in *Through the Looking Glass*, Lewis Carroll explains the meaning of words:

> "I don't know what you mean by 'glory,'" Alice said.
>
> Humpty Dumpty smiled contemptuously. "Of course you don't—till I tell you. I mean 'there's a nice knock-down argument for you.'"
>
> "But 'glory' doesn't mean 'a nice knock-down argument,'" Alice objected.
>
> "When *I* use a word," Humpty Dumpty said, in rather a scornful tone, "it means just what I choose it to mean—neither more nor less."
>
> "The question is," said Alice, "whether you *can* make words mean so many different things."
>
> "The question is," said Humpty Dumpty, "which is to be master—that's all."

This is, of course, just what mathematicians do with language—steal its words for their own purposes—and what programmers do with program variables—make them mean what they want. But we are not used to natural language words being treated that way, and we think as Alice does that words like "glory" can't be messed with. Unfortunately, it is all too human to behave just as Humpty Dumpty does, and take over words arbitrarily. There is even a word for these words of which we are "master:" jargon. In a new discipline, the jargon isn't well established or agreed upon, so the very tool that is best at establishing intellectual control—our language—can betray us. We have already seen this in the lack of agreement about names for the phases of the software life cycle in Section 1.2.7. If two people are trying to talk about software development, and one of them uses "specification" while the other uses "(architectural) design" for the same idea, things quickly get out of

control. In Part IV, something similar will happen with the word *test*. It is used to mean at least these things: an input value that gets tried for a program; a pair of values (input, output) that describe what a program *should* do on one execution; a similar pair that is what the program actually *does* do; a collection of inputs to try, or pairs, etc. So when someone says, "the test failed," it's hard to know just what is meant.

It would be better if it were not so, but in talking about software engineering, watch out for words with shifting, imprecise meanings.

2.2 Divide and Conquer

The only general problem-solving technique for reducing complexity is to break a problem into pieces. Finding the right pieces is an art requiring talent and creativity. Without this talent, two bad things can happen:

Trap 1. Solving the pieces may not solve the problem. It is essential that any decomposition plan include a way to put the pieces together into the desired solution.

Trap 2. Some of the pieces may be as hard to solve as the original problem. A "decomposition" plan that hides essential difficulties in one of the parts, solves the other parts first, then again tries to decompose the difficult part, has merely wasted time by deferring the real problem.

The creative genius to decompose a complex problem so that the complexity is really reduced is art, not engineering technique. But there are a few guidelines to be used in seeking such breakdowns.

2.2.1 Independent Parts

It is the essence of decomposition that the parts must be independent. Intellectual control is not improved unless a person can concentrate on one part without thinking about the others.

2.2.2 7 ± 2 Rule

Human beings seem to operate psychologically according to a "7 ± 2" rule. We keep intellectual control of collections best when they contain five to nine parts. The first rule for divide and conquer is therefore that there should be only a few items in the decomposition. (There are five parts to the software life cycle!)

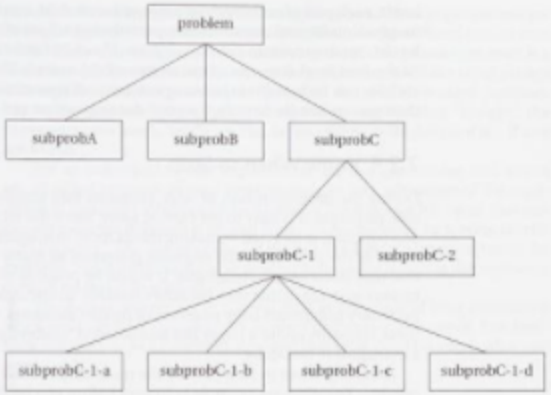


Figure 2.1 A problem decomposition tree.

2.2.3 Hierarchies

Breaking a large, complex problem into about seven parts guarantees that at least one of the parts will also be large and complex—the best one can hope for is to divide the complexity by 7, and if some of the parts are *not* large, others will have to be closer to the difficulty of the whole. To avoid falling into trap 2, it is necessary to continue by decomposing the parts, and the new parts in turn, until *all* parts are small and simple enough that their solution can be glimpsed.

The resulting problem decomposition can be drawn as a hierarchy tree—with the original problem at the root, under it the first-level breakdown, and subsequent breakdowns under each part—the leaves of the tree being problems that are simple enough to require no further decomposition. In this tree, the nodes are the subproblems, and the arcs mean "is a subproblem of." For example, the decomposition tree shown in Figure 2.1 represents a problem broken into three parts: the first and second parts to be solved directly, and the third part further decomposed into two more levels.

Hierarchies have the virtue that they may contain a plethora of parts, yet no part of the structure is very complex. For example, a tree with seven

levels, each part of each level decomposed sevenfold, can meet the criterion for good intellectual control of each part (the branches) and in the whole (the levels), yet may contain $7 \times 7 \times \ldots \times 7 = 7^8 = 5{,}764{,}801$ subproblem leaves (if the root level does not count as one of the seven). That is, our human abilities can be brought to bear on problems millions of times more difficult than we can handle directly, if a good decomposition can be found.

2.2.4 Know When to Stop

During the decomposition of large elements into smaller, more manageable elements, it is easy to get carried away. Since the technique of "divide and conquer" is targeted at making things more manageable, it becomes an abuse of the principle if the engineer generates so many smaller elements that they become unmanageable. It would be possible to generate specifications so detailed that they become unusable. In circumstances where it is necessary to generate large numbers of smaller elements, we must organize these elements under a larger but less detailed "umbrella" in order to have a manageable hierarchy.

Knowing when to stop may be the most difficult task of software engineering. The ultimate example is to stop before you begin, when software engineering itself is not needed for an easy problem. Once engineering is begun in the requirements phase, it is tempting to go beyond system requirements and begin to solve the problems they pose. For example, when discussing database requirements with an end user, an engineer might decide that linked lists are the storage mechanism to use. This decision is inappropriate—it mixes the concerns of the design phase into the requirements discussion where they are not necessary. "Knowing when to stop" can be thought of as making no decisions until absolutely necessary.

2.3 Identify the "Customers"

A basic tenet of producing quality goods of any kind is to keep in mind the needs of customers. In software development, there is a clear "customer" for the software, the user who met with the developer to work out the requirements. The user is also directly involved in the maintenance phase. But in between, in the details of the different life cycle phases, the user is not so clearly the customer. In the testing phase, for example, the engineering goal is to uncover problems in software. Of course, this is done so that the ultimate user will not encounter those problems, but the test engineer is not working only for this user. The test engineer's customer is also the programmer who must repair any mistakes uncovered.

In each of the software phases, it provides clear goals and focus to ask "for whom is this being done?" The answer is always the same. The primary customer for each phase is in the subsequent phase. Thus requirements are "bought" by a developer who writes specifications from them. The developer "sells" these to a designer, whose customer is a coder, whose customer is a tester. In addition, the tester is a secondary customer for all the other phases although it is the test plan that is "bought." Another secondary customer lies in the previous phase (particularly for testing): having "bought" the previous phase work, feedback can be provided on its deficiencies, if any are found.

For an individual developing software, the designer, coder, and others are all rolled into one person. Nevertheless, to take advantage of the separation of concerns inherent in the phases of the software life cycle, that one person assumes different roles and works most efficiently by staying strictly "in character." The designer should not act like a programmer or tester, for example. But remembering who your customers are provides the necessary control on this role-playing.

The rule is: always try to put yourself in the place of your customers: when working for a customer, try to meet that customer's needs. The feedback corollary is: as a customer, demand that work done for you is what you need.

2.4 Fuzzy into Focus

The human mind has a capacity for abstraction that has served the human race very well since its beginnings. Early humans only had to be attacked by carnivores a few times before learning to be cautious around creatures with fangs. The ability to abstract allows humans to live in a world where cars come in different shapes and sizes, radios and televisions have controls in different locations, and stuffed animals with fangs are harmless.

However, though ability to abstract is arguably the human mind's greatest talent, it is also its greatest liability. Generalization leads to stereotyping, which can hinder our problem-solving ability. To illustrate this limiting concept, imagine a door with the door handle placed on the same side as the hinges: people entering the threshold would push against the door handle expecting it to swing out or in, but in fact it would be the *other* side of the door that would open. This would lead to a great deal of surprise, and a phenomenon known as cognitive dissonance (confusion).

In the arena of software development, abstraction has similar strengths and weaknesses. If we share a similar set of experiences, we tend to have similar ideas about the meaning of general concepts. For instance, if a software engineering team has built several payroll systems, they will probably share an idea of what is expected when a customer asks them to deliver "a payroll system." However, if this team is asked to build "a student registration system" (and they have never built such a system), the team members will probably have completely different ideas about what such a system is.

In the course of software development, a vague idea must be translated into thousands of lines of precise code. This work is the opposite of abstraction, and as such, runs contrary to what the mind is inclined to do. In trying to accomplish a task that is not well suited to the nature of the human mind, we find that problems we encounter are directly related to our tendency to abstract: sometimes important details are left out of our specifications and ambiguities creep in.

In conceiving a software system, it is not uncommon for the initial proposal to be a few vague diagrams drawn on a piece of paper. Translating this diagram to a completed, running system is what software development is all about: bringing a "fuzzy" idea into crystal-clear "focus." The ultimate focused system is what we want, but we have to start with the fuzzy diagrams because at the beginning they are the best we can do.

The idea of "fuzzy into focus" applies at all levels of software development. At the beginning, a good software engineer will take the initial diagram and develop clear, concise, and unambiguous specifications. These specifications have brought the initial diagram into better focus, but after all, specifications are not code. During the design phase, the specifications are brought into better focus by developing a general software system architecture to satisfy requirements. Then the requirements are partitioned into the architecture's subsystems; that is, each requirement is assigned to a particular part. More clarity has been introduced because now we know *where* each of the requirements will be fulfilled. Clarification continues until concrete code—that could not be less fuzzy—is reached. The next task is to let software engineers design each subsystem so that implementation engineers can then take the final step of translating the subsystem design into concrete code.

2.5 Document It!

A dangerous myth in software development is that work done well is "self-documenting." What "self-documenting" means to most people is that it is unnecessary to record anything outside the primary product being created. Thus self-documenting code needs no explanation; everything about it can be learned by reading it directly.

If all the products of a life cycle phase have been explicitly identified (as we identified a test plan in the requirements phase even though it is not the requirements document), *and* if no mistakes are ever made, then the "no documentation" of "self-documentation" may be all right. But the existence of a maintenance phase recognizes that there will be changes to software, mistakes at some level that must be corrected. Because of these changes, one of the secondary customers of a phase becomes the very person working on that phase. When change is required, it will be necessary to understand what was done *and why* in order to redo it. The explicit documents of the phase, whose customers are in other phases, are no help with

changes in the phase itself. The most obvious illustration of this is in the coding phase, whose primary product is a precise program expressed in some programming language. Programs are notoriously hard to understand, and without comments describing what they intend, and how they accomplish it, most people (even their authors, when time has passed) simply cannot understand them.

2.5.1 Record It or It's Lost

Our inclusion of a test plan as part of the requirements, specification, design, and coding phases is in recognition of the truth that the most obvious information, once used but not written down, is lost forever. A programmer, coding some difficult algorithm, often "worries" about whether the code written is correct. At the time such a worry appears, the potential problem is glaring, and the programmer would love to conduct a test to see that all is well (or that all's not well, so it can be fixed). But the code is not complete and can't be tested—that lies in the next phase. The test must be recorded, along with the concern that gave rise to it, because weeks and hundreds of hours of programming later, it is almost certain that the programmer will not be able to recall the "worry," much less the specific test that would check it out.

2.5.2 Know What You've Assumed

Assumptions are an important part of developing software because assumptions are useful to constrain a problem to a reasonable and solvable domain. For instance, in writing a registration system for a community college with an enrollment of 2000 students, it is reasonable to assume a maximum capacity of 5000 students for the registration system. But suppose that the college intends (and intended all along) to sell this package to other universities to cover the development costs. If this information comes out only at the end of development, the assumption would be a disaster, since it will have dictated data structure design, storage capacity, system runtimes, and commercial support package selection (e.g., of a database management system).

If assumptions are documented at each stage of development, these assumptions are placed on public display for review by each person who reads the document. If any of the assumptions are incorrect, the odds of early discovery are improved.

2.5.3 Traceability

One documentation trick has proved particularly useful in relating each life cycle phase to the next. *Traceability* means being able to find the parts of the phases that go together. Consider how a software designer works from a

requirements or specification document. Some small part of the document states *what* should be done to handle some special case. The designer of that part of the software is charged with stating *how* it should be done. Once that element of design is included in the whole, it will be hard to reconstruct where it came from and why. But if the parts of the requirements document are labeled, and those labels are carried as documentation on parts of the design, reconstruction will be easy. Similarly, design elements can be traced to the implementing code and to the test plan using the same label.

Any change can have repercussions in every phase of development, can affect all of the documents that exist and all the work that has been done before the change became necessary. Without traceability through requirements, specification, design, coding, and testing, the most minor change can turn into a nightmare of repeated work—work that is compromised because its integrity is lost and produces software no one can trust.

2.5.4 A (Document) Place for Everything and Everything in Its Place

Duplication of information conflicts with updating. If information is kept in more than one place, its copies may be changed to disagree with each other. Duplication of information results from poor organization of a document. The structures of the requirements, specification, and the design, which are repositories for vital information, should lend themselves to the tasks for which these documents will be used. If this structure is clear and concise, there will be one and only one logical and obvious place for each piece of information. Duplication of information can result when an important piece of information is inserted in the wrong place by a well-meaning engineer guessing at where it belongs. When other engineers go looking for this information in the document and can't find it, they may well "rediscover" the information by talking to a user or another engineer. Once they've got the information, they may put it in a different location in the document. Now that the information is in several places, there is a maintenance problem.

It is best to organize documents so that there is an obvious "place for everything," and if information is discovered to be in the wrong place, it should be moved to the correct location to put "everything in its place."

2.5.5 Documentation Isn't a Novel

Good writing style has characteristics that are *not appropriate* in documenting the complex process of software development. Here are three things that most of us like in fiction (and nonfiction), and why they should be avoided in documentation:

The great novels are those that are just slightly obscure so that readers cannot exactly be sure what they mean.

Cleverness. The best writing takes all of our intelligence to figure out—if it's too easy, we find it boring. But in documentation, clarity should come first. If a clever presentation loses even the most plodding reader, the document has failed in its purpose.

Varying terms. It's boring to refer to anything over and over using exactly the same word, and writers are taught to avoid this repetition. But in documentation, the reader is more interested in being sure of the subject than in variety. If it's a "garg," call it a "garg" and nothing else, even if "blorg" is a synonym.

Distributed ideas. Novels don't usually have an index, and if one did, each index entry would have many defining page citations, with information spread throughout the book. Documentation should be just the opposite, with every concept defined in exactly one place. It's hard to read straight through documentation [illegible] things up.

However, most of the rules of good writing apply to documentation. In particular, it's important to use parallel constructions, to use paragraphs appropriately, to keep to the subject, etc.

2.6 Input/Output Is the Essence of Software

The fundamental nature of software is to transform a set of inputs into a set of outputs. This seemingly simple concept (Figure 2.2) is a pervasive notion that needs to be remembered throughout the software life cycle. Much of the early effort in the software life cycle is directed toward:

- Identifying all the outputs from and inputs to the software. Sometimes these inputs and outputs are not apparent because they are hidden from view (e.g., system time and date, databases).
- Identifying the valid and invalid ranges of the inputs and outputs.
- Identifying particularly significant values of inputs and outputs. For example, checks are not issued for 0.00 even though a zero balance may occur.
- Identifying important combinations of the input and output values.

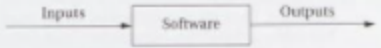


Figure 2.2 Software as a transform.

2.7 Too Much Engineering Is Not a Good Thing

Software engineering is a difficult and challenging profession. It requires discipline, hard work, and technical knowledge. It is a profession that involves constant learning that takes the engineer into fascinating areas of application and often to the leading edge of other technologies such as electronics, mechanical engineering, and mathematics. It is easy to lose sight of the primary purpose of software engineering: to produce software.

2.7.1 If It Ain't Broke Don't Fix It

A system can be "improved" to the point that it no longer works. Here are some examples of engineering a system to death:

- After the system has passed a majority of its tests, and is in the process of undergoing a final release, software engineers are sometimes tempted to completely rewrite a subsystem because it is convoluted or inefficient. This can introduce a whole new set of defects that can delay deployment of the system for a long time.
- The requirements, specification, and design processes are not the end result of the software process. However, the methods and structures that are to be used by these processes can be debated and discussed *at length* by obsessive engineers. This debate can set the project schedule back by an unacceptable margin. The phenomenon is sometimes called *analysis paralysis*.

2.7.2 The "Creeping Feature Creature"

Software developers must remember that the end user is not a software engineer, and is not using the software in a development environment. Instead, the end user may be trying to generate information quickly for a staff report, or generate payroll for a company, or trying to deal with a long line of irritated customers. The end user is supreme when feature sets and usability issues are concerned.

Software engineers are prone to discovery of "necessary," "useful," or "essential" functions at a late stage of software development. Often, engineers will install these features without discussion, hoping to surprise and delight the user. "Creeping features" have been known to destroy projects by delaying the development or deployment of features that the user had specified. The user is surprised by the delay of an essential software system, but seldom delighted. Sometimes extra features are indeed desired by the users, but the impact on delivery date must be presented to the user as an expense of having them installed.

It is not good engineering to deliver a luxury car when the end user wanted a basic truck.

2.8 Expect to Deal with Change

Although lip service is often paid to the desirability of novelty and change in our lives, few people really like upheavals in their world. When something fundamental changes in a software project, it is always a setback because work is wasted and must be redone.

2.8.1 Plan for It

Perhaps it is unnatural to plan for changes—why look for trouble? The question seems unanswerable in daily life, but for software the answer is: because trouble is bound to come, and planning for it can minimize the trauma that changes will cause, allowing them to be made routinely and correctly. Mistakes get made and must be corrected. New information is uncovered, and old assumptions prove false. And in software development, the most unpleasant changes are direct and common: the customer changes his mind.

2.8.2 Fix It Now

Once the need for change is clear, don't procrastinate. The implications of a change will not be known until it is accomplished, so making it should have first priority, however unsettling or difficult that may be.

2.9 Reuse Past Work

There is a large investment in a properly conducted, properly documented software project, in getting it right and keeping it right throughout development. When the project is complete, its organization will be a huge help in maintenance of the software. But a completed project has value beyond the life of its product because parts of it can be *reused*.

2.9.1 Previous Work Well Done Is Golden

One of the great inventions of the world goes under the trade name of Bisquick (Krusteaz is another brand). This stuff looks like white flour, and indeed it is mostly composed of flour, with powdered milk, baking powder, salt, and other ingredients of batters and doughs mixed in. Pulling the Bisquick off the shelf instead of mixing these ingredients from scratch saves a few minutes any time you want to make pancakes or biscuits. The story is that Bisquick was "invented" by someone who realized that if he mixed up a double batch of dry batter ingredients, it took no longer, but half would keep for next time. That's the principle of software reuse, only for software

We freely donate the name "BitQuick" to anyone who wants it. Before you use it, it would be wise to check that someone else hasn't got there first.

"batters," which are more ideas than material stuff. one batch can be used over and over.

Software projects, or parts of projects, repeat themselves. To take a current example, screens full of icons, buttons, and menus occur in almost every PC product, and the requirements, specifications, designs, and even code and test plans for them are more similar than they are different from one another. When developing your twentieth PC product, you've done it all before. It seems a shame to do it all yet again, particularly because it's boring and error-prone, yet since the new product is not exactly the same as the old, you may have to. But if the common ingredients could be identified, they might be adapted to a new project with a great saving of time and a great improvement in quality.

Programmers have shared code from the start of the craft. Indeed, the IBM user organization has always been called SHARE, and it came into existence to pool subroutine libraries.

If code is the most obvious thing to reuse, it is not the only or even the best candidate. Each document starting with requirements may be easier to mine for its content and modify for a new project than it would be to repeat the phase that made the document. In the extreme, activity on a new project would be nothing more than modifying documents of an old one. The gain in time is substantial, and so is the potential for quality. The old project worked in the two senses that it was carried through to completion, and its code runs.

2.9.2 Create Software So that Reuse Is Possible

Software reuse doesn't just happen by looking back at old projects. The customers for reuse are the developers of future projects; unless their needs are thought of, they will find it very tough to get anything from the past. Planning for reuse means always trying to see present work outside the present context. When building part of a widget, an engineer is thinking, if only unconsciously, "what does a widget need here?" The (again, unconscious) rationale for a choice might be: "That's not important, widgets don't ever do that." In contrast, reuse thinking is: "What might get in the way of using this when it's *not* a widget, doing the very things widgets don't do?"

Always steal code instead of writing it anew. But don't steal the wrong code, and don't steal code that doesn't work.

Generality is important in making software parts that can be reused, but so is granularity and encapsulation. Chances are that what's needed in the future will be like a small part of what's being done now, not so much like the whole. The parts must not be inextricably woven together, and they must be small enough to recognize.

The final issue is one of quality. Work intended for reuse, since it may be placed in a context that is not imaginable when it was created, must be of much higher quality than work that is done quickly, fiddled with until it can be used in a limited context, and then discarded.

2.9.3 Don't Reinvent the Wheel

"We could get those matrix routines from Stewart's textbook." "Nah, I'll write them myself."

In medical research circles, the acronym NIH stands for the National Institutes of Health. In software development, it stands for a syndrome that is anything but healthy: Not Invented Here. What's wrong with someone else's work? Maybe nothing, just NIH.

Software people, since the day the first program was written, have felt that it is better to be in complete control, to "do it myself." There's some justification for this attitude—programmers who have tried to use someone else's code and found it to be riddled with subtle bugs, are determined not to be burned again in the same way. But this attitude is easily carried too far. In the 1960s, a lot was learned about operating systems for large mainframe computers. When minicomputers arrived in the 1970s, their operating systems were mostly reinvented from scratch. Their designers didn't read about the earlier work. In some cases, early lessons were embodied by programmers who worked just down the hall, as when the DEC PDP-11 systems group did not talk to the DEC PDP-10 systems group. When the microprocessor arrived in the 1980s, again its operating systems were invented from scratch. All this reinvention accounts for Microsoft systems that are far inferior to mainframe work done 40 years ago. The hardware designers did not make the same mistake: the CPU chip that those PC systems run on can do all the tricks that the 1960s' mainframes did (and do them far cheaper).

In other engineering disciplines, the first stop on a new project is the library, to see what's been learned in the past. A lot of education is "reading the classics," from Dickens for budding novelists, to ancient Roman highway construction for civil engineers. Software engineering has a long way to go.

2.10 Take Responsibility

Systematic procedures for getting things done, separating the concerns in different parts of the process, can be an excuse for blaming and finger-pointing instead of dealing with difficulties. For example, if during coding, an ambiguity in the design is encountered, the coder can think, "not my problem" and largely ignore what the designer failed to do. And if it manifestly was the coder's problem, because that same person did the design, then "can't deal with that now, fix it later," is the same kind of thinking. What the developer has to keep in mind during all phases of the development process is that he or she is directly and unremittingly responsible for the software. A myriad of things can go wrong, and if everyone involved has the attitude that any difficulty has to be examined with care and dealt with properly, then there is just a chance that all errors will be caught and the software will work properly.

In almost every kind of engineering, construction includes "safety factors," to account for unexpected or extreme conditions. For example, the girders of a bridge are made a bit stronger than calculations indicate is

needed, just in case. Software engineers could use ideas like redundant code verifying a calculation, checksums, and other data-structure integrity mechanisms to try to protect against their own mistakes, but except in extremely critical applications, they do not use these devices. The reason probably lies in the myth that software is "logical" rather than physical, and hence can be perfect, not affected by forces of nature. It has long been known that human mistakes in software development are just as damaging (and occur far more frequently) than "acts of God" that bring down the structures of civil engineers. But for the foreseeable future, software safety rests with the care of human beings doing their best.

Procedures for developing software, like laws, have a "letter" and a "spirit." And like laws, the letter very imperfectly captures the spirit. Thus the people involved have to look beyond what they are "supposed to be doing" to what they are "supposed to achieve," and recognize that when the two don't match, it is the outcome, not the rule, that matters.

Flaws in all software have their source in the development process. To have seen the process going wrong and done nothing is irresponsible.

2.11 Summary of Software Engineering Principles

The subsection titles of this chapter can be woven together into advice for the software engineer:

> Take responsibility for the software you produce. Identify your customer and your customer's needs, and think of them in input/output terms. Research what has been done before. Stay in intellectual control of your efforts, and document what you do. When a problem is too difficult, divide it, always striving to bring more focus to initially fuzzy solutions. And when you've done everything perfectly, there will be changes, so be ready for them.

These principles will find application throughout this book.

How Does It Fit?

1. Group the principles of this chapter under three headings: (a) Of primary importance, (b) moderately important, and (c) peripherally important. List about the same number of principles in each group, and give a brief justification for putting each item in groups (a) and (c).
2. Pick two of the principles in this chapter. Give an example of each one, an example as far removed from software development as you can invent.

From section 2.1

3. The following formula was seen on a blackboard in the physics building of Cornell University in the 1960s:

$$\lim_{5 \to 4} \sqrt{3} = 2.$$

Relate this to Humpty Dumpty's idea about the meaning of words.

From section 2.2

4. Give an example from your own experience (not from software development if possible) of falling into divide-and-conquer trap 1.
5. Explain how a problem decomposition described by a sevenfold branching tree with seven levels after the root differs from decomposing the same problem into 7^6 arbitrary parts. How would you draw a tree for the latter decomposition?

From section 2.3

6. Explain why too much thinking about "customers," and what they want and need, can violate the principle of separation of concerns.

From section 2.4

7. What excuse can be given for bringing "fuzzy into focus" as opposed to working harder and never being "fuzzy" at all?

From section 2.5

8. Some people have an amazing memory for detail. They can recall the characters and plot of every novel they've read, for example, or rattle off textbook lists they mastered years ago. Suppose such a person is developing a piece of software. Does that person need documentation? Why or why not?
9. It has often been said that a writer (like a novelist or poet) must write for himself, never for his public. Why is this, and why doesn't the advice apply to software engineers writing code? Are there situations in which this advice doesn't apply to writers? Are there special software engineering situations to which it does apply?

From section 2.9

10. Comment on how hard it would be to steal code from an existing project as a function of its size, and on the worth of the effort. For subroutines of a few lines of code, it would be much easier to write them than to find, read, and understand existing ones. At the other extreme, it seems just as difficult to reuse a 100,000-line program than to write it. Where is the point at which reuse is worthwhile?

Review Questions

1. List the major principles described in this chapter and explain each briefly in your own words.

Further Reading

The authors of this text are not alone in collecting principles of software engineering. We have more than Boehm [Boe83] but fewer than Davis [Dav95].

The original work on "7 ± 2" appears in one of those articles that is worth reading just because everyone refers to it, but most have not read it [Mil56].

As an example of system requirements "beyond the state of the art," the so-called star wars system is considered in detail by Dave Parnas [Par85].

Anyone interested in writing of any kind, must read "the little book" [SW59], the collection of dogmatic grammar rules from William Strunk, Jr., and the wise thoughts of E. B. White on writing well.

References

[Boe83] Barry W. Boehm. Seven basic principles of software engineering. *The Journal of Systems and Software*, 3:3–24, 1983.

[Dav95] Alan M. Davis. *201 Principles of Software Development*. McGraw-Hill, 1995.

[Mil56] J. Miller. The magical number seven, plus-or-minus two: Some limits on our capacity for processing information. *Psychological Rev.*, pages 81–97, 1956.

[Par85] David L. Parnas. Software aspects of strategic defense systems. Technical report, University of Victoria, July 1985. DCS-47-IR.

[SW59] William Strunk, Jr., and E. B. White. *The Elements of Style*. Macmillan, 1959.

chapter 3

Is It Really Engineering?

Much of the human activity of dealing with the objects in the world can be conveniently categorized as:

Art. The production or expression of what is beautiful, appealing, or of more than ordinary significance.

Craft. A trade or occupation requiring special skill, especially manual skill. Carries the shade of meaning that this skill is applied to bad purposes, as in "crafty devil."

Science. The systematic study of man and his environment based on the deductions and inferences that can be made, and the general laws that can be formulated, from reproducible observations and measurements of events and parameters within the universe.

Engineering. The art or science of making practical application of the knowledge of pure sciences; a profession.

(The definitions are taken, somewhat edited, from *The Macquarie Dictionary*. Most of the definitions include each other as special meanings. For example, *Art* is also defined to be "craft" or "science.")

Where does the making of software fall in this categorization?

Science seems the least appropriate since observations and measurements, not to speak of laws, are involved at most peripherally. *Art* may sometimes apply, but most software could hardly be called beautiful. Probably the best case today can be made for *craft* more than for *engineering*. Both are occupations of making things, but the engineer is a professional who applies scientific knowledge. This book is about the knowledge required to engineer software. (On *engineer* as a verb, the dictionary is a little more lenient: "to arrange or carry through by skillful or artful contrivance.") The reader must judge whether there is enough science here to take us beyond *craft*.

If program developers are to be "software engineers" and their activities are to be "software engineering," then it is worthwhile comparing the field to

older, more established branches of engineering. In this comparison, there are three aspects to engineering:

1. The technical and scientific definition of the field.
2. The professional, organizational side.
3. The social responsibilities of the profession.

3.1 What's Different About Software?

Software development is different from other kinds of engineering because:

1. Computer hardware costs (almost) nothing and software can do (almost) anything. As a single example, a complete microprocessor can be installed in an appliance like a washing machine for roughly the same price as a conventional electromechanical switch, but the microprocessor can be used to make the washing machine appear extremely "smart" in its operation, and it will probably far outlast the switch.
2. Software is backed by "science" more like mathematics than like physics or chemistry. That is, it is artificial and logical, not tied to the physical world and its laws of nature. People invented both computers and their programs, and their creations are arbitrary. (Hardware operates within natural laws, but it is a good first approximation to assume that hardware answers only to an arbitrary, human design.)
3. Software is perhaps the most complex of human artifacts, both in itself and in the process that creates it; there is wide scope for human error.
4. Software development operates at a considerable remove from software use, so the consequences of developer actions are difficult or impossible to foresee. The more general a software solution is, the more likely that it will be used in ways the developers did not imagine.
5. The software industry is very young, and its explosive growth is unprecedented; hence many of the usual societal forces have not yet come to bear on it. At the same time, since software is growing ever more important in daily life, ethical issues do not wait for these institutions.

These differences make software very important in the world, yet not very well understood. The tried-and-true ideas in other branches of engineering may not work for software.

3.2 Artificial Science

In Einstein's famous statement, he did not believe that God was "damn mean," only subtle in framing the laws of physics.

The "scientific method" has as its fundamental idea a kind of contest between nature (or God for those who believe that nature is God's creation) and the human mind. The contest is an unequal one (even for those who do not believe in God) because nature is always right, and humans are usually wrong. In the contest, people try to figure out how the world works, according to some assumed rules. One of the rules is that nature plays fair by invariably repeating itself. The results of observations, if made carefully enough, can be trusted to be repeatable. Another rule is that the natural system is elegant and simple so that people have at least a chance of figuring it out. In the contest, people develop a theory to explain the mechanism by which a natural phenomenon operates. From this theory deductions are made about details of the phenomenon. An experiment is then performed to observe the details, and if it agrees with the predictions, the theory is "supported" (but theories are never "proved"). If the experiment disagrees with the theory, the theory is disproved.

Engineering is based on science and constrained by it. The engineer tries to make things work but is unable to break natural laws. Thus, for example, theoretical physics has long supported (and experimental physics so far not disproved) the law of the conservation of energy. The mechanical engineer working within this law does not therefore expect to get something for nothing, like vehicles that require no fuel, or perpetual motion machines.

Software is in no sense a natural phenomenon. It has no laws but accidental ones that its human creators happened not to violate—and these are seldom elegant. Like mathematics, its "science" is no more than logical implication. As a mathematical example, if a real-valued function is continuous, then it has a Riemann integral. This theorem can be deduced from the definitions of the terms in it, and no one expects that it can possibly go wrong (say, by the discovery in Australia of some particularly nasty continuous real-valued function). In just the same way, if a program of length n is a member of a context-free language L, then it can be parsed in time $O(n^3)$ by a parser that properly recognizes all and only the programs of L. No one believes that counterexamples will be discovered on the hard disk of some PC in Tasmania. There might be a mistake in the proof of the theorem about Riemann integrals. There might be a mistake in a parser for L. Particular examples of functions and programs (respectively) might expose such mistakes. But the role of "experiments" with examples like these is completely unlike experimental physical science. In science, an experiment might disprove a theory and the theory would have to be discarded; in computer science, such an experiment serves only to expose a "bug" in human logical work. We continue to hold an unswerving belief in the theorem or the parser, which we fix up to handle the examples.

It is perhaps the most interesting thing about mathematics (and software) that despite its artificial, human-defined nature, it is subject to limitations. The software ones are better known than the mathematical ones (although they are of much the same character). For example, it is impossible, *in principle*, to find an algorithm to determine whether an arbitrary piece of software will ever abort with a runtime error, even if the particular input on which the software begins is given. (And this "runtime error problem" is of considerable interest to software engineers, who would certainly like to know if their programs are going to abort.)

These philosophical considerations have a profound effect on the way software research is done, and on the intellectual discipline of computer science. But most of the practical aspects (the *engineering* aspects) of software development arise in treating software as just another commercial product.

3.3 The Analogy to Other Professions

In other professional fields, tangible products are developed, and analogies exist between these products and pieces of software. The analogies are just that, though perhaps as often misleading as helpful. When information about software development is presented in the rest of this book, the reader may find it interesting to think about whether software is like a skyscraper, or an airplane, or a car, or a computer chip, or something else, and may want to look for lessons from the other fields.

3.3.1 Civil Engineering

Civil engineers design buildings and other large structures like bridges and dams. (The name of their discipline arose when it was necessary to distinguish them from the *military* engineers who do similar things for a different purpose. The U.S. Army Corps of Engineers is the best known modern group of military engineers.) The safety of these structures is of paramount importance. The science behind civil engineering is strength of materials, mechanics and hydraulics, on the grand scale referred to as "the forces of nature" (or "acts of God"). Newtonian physics provides elegant and accurate solutions to equations that describe how (say) a bridge will behave, and standardized components (like steel I-beams) are manufactured with well-understood qualities. The civil engineer can therefore predict how a design will behave when constructed and put out in the world; by adding "safety factors" that make the structure even stronger, a remarkable success record has been accumulated. (Not that there have not been also some spectacular failures, which have led to changes in materials and designs.)

3.3.2 Architecture

A civil engineer designs buildings only at a detailed level; it is an architect who does the high-level design. Architects are supposed to know enough science that they do not design buildings that are impossible to build safely, but their business is primarily fitting the building to its intended use and satisfying its human owners and occupants. Thus architecture is often taken as the model for the most difficult and creative part of software development, the overall concept within which a program is developed. Software engineers have even appropriated the name, calling the crucial first design ideas "software architecture."

3.3.3 Mechanical/Aeronautical Engineering

The industrial revolution of the 19th century was built on the application of the power sources of first, falling water, and then, steam. The engines drove all manner of gadgets that helped to make textiles and other consumer goods. Today, design of cars and airplanes is the apex of the mechanical engineering business. Both are relatively complex systems of cooperating parts, subject to wear and tear as well as to flaws in design or imperfect materials. It is said that a present-day commercial airplane cannot fly until the weight of paper design documentation is equal to the weight of the plane. "Maintenance" is a necessary part of the continued functioning of mechanical devices, including carefully conducted inspections to find and replace what is beginning to wear out. The components of cars and planes are assembled in hierarchies, where a component is built from other components, and in turn may be part of a larger component. Making these components work reliably together is a large part of design.

3.3.4 Computer (Hardware) Engineering

This subdiscipline of electrical engineering is the envy of all other fields today. The cost of computers has fallen repeatedly by orders of magnitude, whereas the capabilities and desirable characteristics (like speed, small size, and low power consumption) have improved even more dramatically. This progress is the result of massive integration of parts. Where the first computers were giants wired up by hand, a modern PC (which costs 1000 times less, and has 10000 times the speed) is built from a handful of semiconductor chips, each performing the functions of hundreds of thousands of separate parts. Thus the complexity of a computer is in its chipset, and chip design is the heart of computer engineering. A modern chip could not be designed without computer assistance, in the form of CAD (computer-aided design) programs. These help the designer lay out the circuits that will go on the chip, and the manufacturing process is heavily automated once the rather abstract design has been built in a CAD workstation.

The analogy to hardware engineering is naturally the most appealing for software, and has even given rise to something called CASE (computer-aided software engineering). But the dangers of analogy are also most evident here. Hardware at its most complex only begins to approach real software complexity. More important, hardware has constraints on its form that originate in the physics and materials science of its manufacture. These constraints paradoxically allow hardware designs to be under better human control. Software lacks all constraints, which tempts its human designers to construct software that is so crazily structured that it cannot work. The CAD program disciplines the hardware designer to stick to the tried-and-true; the CASE program can be only cosmetic help to the software designer, who is not prevented from trying crazy ideas.

3.3.5 Medicine

Medical doctors do not have a "product," at least not at the present stage of genetic knowledge, but they do diagnose and "repair" health problems that their patients experience, just as software problems sometimes fall to software engineers. The suggestive aspect of the medical analogy to software is that human medical problems are very complex and so variable that there seems to be only a little solid science behind treatment. Furthermore, doctors are consummate "professionals," and have been highly successful at controlling entrance to the profession and maintaining their incomes at a high level.

3.4 Responsibility of Software Developers

If software is a product made and sold like any other, most of the issues of engineering such products are well explored, if not always simple and clear. There is general agreement that anyone who sells a product bears some responsibility for it. The degree of responsibility turns on two points:

- Did the product do significant harm?
- Was its seller negligent?

It is agreed that real responsibility exists only when a user is seriously injured (otherwise nothing of consequence has occurred), and when the producer could and should have done better, but did not. The ambiguity in such a standard leaves wide latitude for interpretation, and the legal system exists to try to fill in the gaps.

A peculiar culture has grown up around software, perhaps as a consequence of its intangible, logical nature. Where software is involved, none of the normal human rules seems to have much force. Thus students who would never think of stealing a book will steal computer files and think nothing of it. People who would think it very wrong to read a letter addressed to a friend will snoop on the friend's e-mail without compunction. And people who would feel tremendous guilt over building a bookcase that falls down and hurts a friend have no problem giving someone software that crashes and causes considerable harm. The sociology of computing is just beginning to be studied, and such phenomena are striking but not very well understood. For software professionals, responsibility for the programs they develop is not an option, but an absolute necessity.

3.5 Engineering Institutions

Society has a need to control the interaction between people who produce a product and those who use it. A number of institutions exist for this purpose and have been used to good effect in a wide variety of circumstances.

3.5.1 Market Forces

People supposedly will not buy shoddy goods if they have a choice, and in a free market system, they should be able to choose a quality product and drive a dishonest producer out of business. The matter is complicated by price—people *will* buy shoddy goods if they are cheap enough, and will put up with them because they cannot afford better. In software, another complication is that users are not very good at evaluation—the goods may appear different than they are because software lacks the usual self-evident marks of quality. Further, software is quite discontinuous from one product (or release) to another, and users have no way of evaluating the quality of changes. Release 5.1 of what was an excellent program at 5.0, although it promises wonderful improvements, may turn out to be terrible.

Special commercial organizations exist to help sort out good products from bad (or safe from unsafe)—perhaps the best known in the United States is the Underwriters Laboratories (UL) dealing with electrical devices. For a fee, these organizations put their stamp of approval on a product after testing it to determine whether appropriate standards are met. However, it is a lot easier to certify a toaster than a word processing program, and although UL is beginning to venture into software, it may be difficult for any organization to acquire the public trust that is essential to such an operation.

3.5.2 The Legal System

Although ultimately it is the body of case law that determines the official societal position on any dispute about responsibility, the law moves

very slowly, and works poorly in complicated situations. If professionals can't agree on what's right, how can they expect judges and juries to do better?

Existing case law applies to software in two quite different ways, depending on the view taken of the software itself. The precedents are in manufactured goods and informational services. If software is taken to be a product, comparable to a toaster, its sale carries clear responsibility for the manufacturer and sometimes for the seller. Should a customer be harmed and blame it on the software, a suit may be settled in favor of the customer, particularly if it is demonstrated that manufacturer or seller did not use proper care. On the other hand, if the software is taken to be an informational service, like a book, then the customer is expected to exercise common sense in taking "advice" from it, and if something goes wrong, cannot blame the maker or seller. "Strict liability in tort," although seldom applied to software, holds the maker responsible without excuse, providing that harm has been done.

3.5.3 Warranties

WE PROMISE NOTHING. YOU ARE ON YOUR OWN!

A warranty is an attempt to turn responsibility into a contract between producer and user, a contract that is part of the sales transaction. The producer agrees to some limited form of responsibility, and in return the user agrees not to hold the producer responsible beyond the limitation. Perhaps the "vacuous warranties" that we joke about in software are really producer attempts to realize the limitation aspect without giving anything in return. However, like all legal documents, warranties are not absolute. In the case of many products, producers have been held responsible even when the warranty explicitly denies responsibility. (In a similar way, many legal-appearing "releases" that we are all asked to sign have no real force if something bad enough happens or there is negligence.) Warranties also have a market function, in that users may be willing to pay for them, or to choose between competing products on the basis of warranties. Warranties provide a way for producers to add value to a so-so product by offering to replace it on warranty. Such a product may then successfully compete with one of better quality, its warranty making it equal.

Software warranties are more problematic than for most products because software does not wear out. An offer to replace software, for example, is not meaningful if the replacement is a copy with the same defect, or a new release with different unknown defects. To make a software warranty valuable, the producer would have to promise to fix problems. The complexity of software makes it doubtful that any meaningful warranty is believable. The producer probably cannot tell if software is adequate, and if it is not, may not be able to fix it.

3.5.4 Professional Organizations

Beginning with the craft guilds of the Middle Ages, the producers of any product or service have had an interest in controlling their own industry. The most important part of control is access to the profession itself. Professional organizations try to restrict the supply of people in their occupation, and thereby influence the price of their services. The American Medical Association is certainly the prime example of success in this control, with the American Bar Association a close second. Second, professional organizations try to police their membership. If some members behave irresponsibly, all those in the profession can be harmed. Professions are partly distinguished by having a generally accepted code of ethics to which their members subscribe willingly. However, it can happen that such codes are adopted more to avoid outside interference than because the "profession" truly accepts them. The dreaded outside interference usually takes the form of government regulation.

ACM officials have several times tried to change the organization's name. The membership, however, always votes them down.

The software profession (if it is one) has two organizations. One, the Association for Computing Machinery (ACM), was formed just after the Second World War. The second organization is a subgroup of the Institute of Electrical and Electronic Engineers (IEEE), the IEEE Computer Society. The Computer Society is probably more concerned with hardware than is the ACM, but both groups have strong software, and software engineering, components.

3.5.5 Government Regulation

A socialist would say that market forces *usually* run counter to the public interest and that the role of government should be much wider.

It is recognized that market forces can run counter to the public good, and that in such cases, government is the only entity with a broad enough scope to control abuses. A case in which this is clearly so is when a government grants a monopoly. Since it has by fiat removed the controlling force of competition, it is beholden to exercise control. Telephone and power companies are monopolies regulated by U.S. state and federal governments, which must approve substantial service changes and rates. Today's telephone companies are less monopolistic and less regulated than in the past, and the same "deregulation" is being applied to power. (What remains regulated are the hardwired distribution systems, since it is in the public interest not to duplicate them.) It is probably too soon to say whether deregulation itself is actually in the public interest. Another clear case for regulation occurs when it is demonstrated that competition is a force in the wrong direction—producers are led to make more and more dangerous products because they are in demand; government then steps in to protect the public (from itself!). Most drug laws, from the federal approval of over-the-counter products to criminalization of "recreational" drugs, are of this kind.

Governments generally do not approve of monopolies that they have not authorized and do not regulate. If commercial interests band together to control the market (or if one dominant company controls it by default), then the government may institute "antitrust" action to break up the "trust" organization and force competition. The theory of the "free" market has it that competition will lower the price and improve the service, which governments are supposed to favor for their citizens (and for themselves as buyers). The most famous antitrust action of recent years resulted in the breakup of the Bell Telephone empire of American Telephone and Telegraph. The U.S. Justice Department is currently following the same course with Microsoft Corporation, claiming that they have used their position as the de-facto supplier of most of the world's operating systems to unfairly compete with other software companies (notably, that Microsoft has conspired to deny hardware vendors their operating system unless they also accepted other software with it, to the detriment of companies trying to independently market such other software). A computer professional, listening to the technical claims being made in court in the Microsoft case, cannot help wondering if the legal system has any hope of coping with software issues.

There are at present only a few beginning attempts to regulate software, and these are part of ongoing regulatory activities. For example, the Federal Aviation Authority (FAA) necessarily deals with software because it is used in the aircraft they must approve for commercial use. The FAA's task is becoming much more difficult because the next generation of aircraft, beginning with the Airbus 320 and the Boeing 777, cannot fly without computer assistance, so the software *must* work. The Nuclear Regulatory Commission (NRC) is beginning to deal with nuclear power plant shutdown and control systems based on software, and hence to certify this software, as they have in the past certified other aspects of plant construction and operation. The Food and Drug Administration (FDA) is beginning to assume responsibility for medical products, both those used in treatment (like X-ray machines) and those prescribed (like heart pacemakers). These regulatory authorities have a far more difficult time dealing with software than with the other aspects of their domains because there is no agreement within the profession on standards and practices.

3.5.6 Professional Engineers and Accreditation

The engineering professions are unique in that their practitioners daily attempt public works that risk a vast liability should they fail. Civil engineers carry the greatest burden because the structures they design could potentially injure tens of thousands of people. To deal with this problem, a joint effort of government and professional engineering organizations has created the position of professional engineer (PE). To earn the PE title, an engineer has to be certified by government (a state in the United States). The certification involves training (approved by the professional association), work experience, and examinations designed by the professional association and administered by the government. On any substantial project, there must be a responsible PE, who signs off on each decision of the project. He is literally responsible because he can expect to lose his livelihood if anything subsequently goes wrong that can be traced to his lack of skill or foresight. Furthermore, engineering schools are accredited by a professional organization, and some projects will hire only graduates of an accredited program.

There have been a few attempts to introduce these engineering institutions into the software industry, but most have foundered on the difficulty of defining what a "software engineer" should know, and how to define the limitations of responsibility for a "software PE." There is a computer science accrediting body, the Computer Science Accreditation Board (CSAB), but its findings do not carry the weight of the ABET (Accreditation Board for Engineering and Technology) organization in the older engineering disciplines. It would be unthinkable for (say) an electrical engineering department at a major university to lose its ABET accreditation, but most of the top-ranked computer science departments disdain to even apply for CSAB accreditation. For the future, college programs explicitly in software engineering (as opposed to more general business or scientific computing) are coming into existence, and it may be appropriate for ABET to accredit such programs. The ABET and CSAB organizations are also talking about a possible future merger.

3.5.7 "Software Engineers" Are Legal Only in Texas

The differences between (say) mechanical engineers who design automobiles and "software engineers" who design programs are substantial, but there are similarities, too. The decision about whether there really is a profession called software engineering, however, is not at all in doubt. Engineering is a well-established profession, and like all professions, its members have taken steps to protect themselves from those who are "not in the club." In the United States (state by state) and in many other countries, laws regulate the use of the title "engineer" after a person's name in a business context. With some exceptions (people who drive trains and members of the U.S. Army Corps of Engineers, for example), these laws prohibit the use of "engineer" as a title unless the person has an appropriate degree from an ABET-accredited department and has passed a state licensing examination. The recognized fields of engineering are precisely defined, and though "sewage" is one of them, "software" is not. According to the legal definition, there can be no software engineers, and people have been prosecuted for

There are 470,000 electrical and computer engineers, but 860,000 programmers; there are 1.3 million engineers of all kinds. (U.S. Bureau of Labor Statistics, 1998)

calling themselves by that title. The state of Texas has, however, begun the first attempt to license software engineers. They began with the legal end of the stick, establishing the license. Now they have only to decide who is allowed to have that license and how that will be determined.

There are reasons why there may never be widespread recognition of "software engineering," not the least of which is far too many people are needed to practice in this field. Professions are by definition small, elite groups. Be that as it may, it is our contention that the ideals, standards, and technical character of software development can be modeled on the engineering profession, and we will use the words *software engineering* proudly, without the quotation marks, in the rest of this book. Let Texas do what it can.

How Does It Fit?

From section 3.1

1. In a discussion with a "real" engineer, say, in electrical power, it may carry the day to describe the way in which shrink-wrap software is sold and serviced—the customers are expected to pay for the developer's mistakes. The power engineer can appreciate the difference. Write a description under the title, "How it would be if toasters were like PC software," describing what would happen if a person bought a toaster that worked as badly as some software works, and tried to take it back to the store.

From section 3.2

2. From your experience in college courses, argue that mathematics, although it's been called the "queen of the sciences," is not a science at all. How do you suppose it got to be "queen"?
3. Building a perpetual motion machine, and finding an algorithm to determine whether software will abort at runtime, are both "impossible" problems. Identify as many differences between them as you can. In particular, if you had to choose which problem is not really impossible, and will someday be solved, which one would you pick, and why?

From section 3.3

4. What is the connection between salaries in a profession and its professional association controlling the number of people who are qualified to join the profession?
5. What if a software engineer behaved in professional life as a doctor behaves? What differences might there be in how the engineer would act? Describe some of the things a doctor-like engineer would and would not do.

From section 3.4

6. For each of the following situations, give a computer analogy, and decide the ethical thing to do, in the situation as described and in the analogous computer situation. If you wouldn't do the same thing, why not?
   (a) On a friend's desk you happen to see a receipt for a piece of jewelry, which you know was a present for a mutual friend. Do you look at the amount and tell the friend?
   (b) First-class mail addressed to someone you don't know is delivered to your house by mistake. Do you open it? Do you see to it that the letter is redirected?
   (c) In a restaurant you often frequent, you notice a mistake on the bill, in your favor. Do you call the waiter's attention to it and pay the extra?
   (d) A friend keeps his garage locked with a padlock. You are standing next to him when he opens it. Do you observe and remember the combination? Do you later go and explore the garage by yourself?

From section 3.5

7. Which of the institutions of this section is the strongest in controlling the harm that can be done by bad software? Answer for:
   (a) As things stand today
   (b) On the basis of potential for the future
   Justify your answers.
8. Do you think that a professional organization like the ACM should take a position on whether software engineers should develop software as their customers require, or sometimes refuse to do so because the software will cause harm? Justify your position in terms of:
   (a) What is good for the profession
   (b) What is good for society
9. If there are too many software developers to form an engineering profession, perhaps they could form a trade union. Discuss whether or not this would be a good idea.

Review Questions

1. Explain why software development is neither science nor art according to dictionary definitions.
2. What institutions are characteristic of an engineering profession? List them in what you consider to be the order of importance for "professionalism."

Further Reading

Philosophers of science have not yet begun to consider computer science, which is a discipline quite distinct from both the physical and social sciences. An exception is Herbert Simon's small book *The Sciences of the Artificial* [Sim96].

Harry Petrosky is a civil engineer very devoted to his profession and aware of its limitations as well as its joys. His *To Engineer Is Human* [Pet85] should be read by anyone aspiring to engineering, but particularly by budding software engineers. Nancy Leveson, an expert in the safety of systems that include software, has written a clever comparison between the dangers of exploding steam boilers in the industrial revolution and the dangers of software [Lev92].

Using any Web search engine, you can discover the site for a collection of reports called "Computer risks to the public," edited by Peter Neumann under the name "RISKS." The site has a good search engine of its own and can be used to explore detailed information on every sort of software and hardware problem imaginable, from malfunctioning of ATMs to airliner crashes. The information on the Ariane-5 rocket disaster in Chapter 10 came from the RISKS site.

Oliver Wendell Holmes wrote a famous poem about a perfect feat of mechanical engineering and what happened to it, "The Wonderful One-hoss Shay" [Hol76].

The ACM has an official code of ethics, which it has published in its membership magazine [ACM93]. Both ACM and IEEE routinely report on the evolving body of software law. A survey of basic legal issues and terms appears in reference [Sam93].

The U.S. Department of Justice is charged with bringing antitrust proceedings against corporations it judges to have monopolies not in the public interest. Its most famous recent case resulted in a court ordering the breakup of the Bell Telephone empire. The DoJ is now proceeding against Microsoft, with an outcome yet to be determined. However, at the same time, almost without publicity, a far-ranging revision of the Uniform Commercial Code (UCC), which governs the legal responsibilities of sellers, is in progress. Cem Kaner believes that the revision is a disaster, and it's hard not to agree with him. The Web address is `www.badsoftware.com`, where you will find links about the UCC.

References

[ACM93] ACM. ACM code of ethics and professional conduct. *Communications of the ACM*, pages 100–105, February 1993.

[Hol76] Oliver Wendell Holmes. The deacon's masterpiece, or the wonderful one-hoss shay. In Richard Ellmann, editor, *The Oxford Book of American Verse*. Oxford University Press, 1976.

[Lev92] Nancy Leveson. High-pressure steam engines and computer software. In *Proc. 14th Int. Conf. on Software Engineering*, 1992. Melbourne.

[Pet85] H. Petroski. *To Engineer Is Human: The Role of Failure in Successful Design*. St. Martin's Press, New York, NY, 1985.

[Sam93] Pamela Samuelson. Liability for defective electronic information. *Communications of the ACM*, pages 21–26, January 1993.

[Sim96] Herbert Simon. *The Sciences of the Artificial*. MIT Press, 1996.