Chapter 1

The Scope of Software Engineering

Learning Objectives

After studying this chapter, you should be able to

- Define what is meant by software engineering.
- Describe the classical software engineering life-cycle model.
- Explain why the object-oriented paradigm is now so widely accepted.
- Discuss the implications of the various aspects of software engineering.
- Distinguish between the classical and modern views of maintenance.
- Discuss the importance of continual planning, testing, and documentation.
- Appreciate the importance of adhering to a code of ethics.

A well-known story tells of an executive who received a computer-generated bill for $0.00. After having a good laugh with friends about "idiot computers," the executive tossed the bill away. A month later, a similar bill arrived, this time marked 30 days. Then came the third bill. The fourth bill arrived a month later, accompanied by a message hinting at possible legal action if the bill for $0.00 was not paid at once.

The fifth bill, marked 120 days, did not hint at anything—the message was rude and forthright, threatening all manner of legal actions if the bill was not immediately paid. Fearful of his organization's credit rating in the hands of this maniacal machine, the executive called an acquaintance who was a software engineer and related the whole sorry story. Trying not to laugh, the software engineer told the executive to mail a check for $0.00. This had the desired effect, and a receipt for $0.00 was received a few days later. The executive meticulously filed it away in case at some future date the computer might allege that $0.00 was still owed.

This well-known story has a less well-known sequel. A few days later, the executive was summoned by his bank manager. The banker held up a check and asked, "Is this your check?"

The executive agreed that it was.

"Would you mind telling me why you wrote a check for $0.00?" asked the banker.

So the whole story was retold. When the executive had finished, the banker turned to him and she quietly asked, "Have you any idea what your check for $0.00 did to *our* computer system?"

A computer professional can laugh at this story, albeit somewhat nervously. After all, every one of us has designed or implemented a product that, in its original form, would have resulted in the equivalent of sending dunning letters for $0.00. Up to now, we have always caught this sort of fault during testing. But our laughter has a hollow ring to it, because at the back of our minds is the fear that someday we will not detect the fault before the product is delivered to the customer.

A decidedly less humorous software fault was detected on November 9, 1979. The Strategic Air Command had an alert scramble when the worldwide military command and control system (WWMCCS) computer network reported that the Soviet Union had launched missiles aimed toward the United States [Neumann, 1980]. What actually happened was that a simulated attack was interpreted as the real thing, just as in the movie *WarGames* some 5 years later. Although the U.S. Department of Defense understandably has not given details about the precise mechanism by which test data were taken for actual data, it seems reasonable to ascribe the problem to a software fault. Either the system as a whole was not designed to differentiate between simulations and reality or the user interface did not include the necessary checks for ensuring that end users of the system would be able to distinguish fact from fiction. In other words, a software fault, if indeed the problem was caused by software, could have brought civilization as we know it to an unpleasant and abrupt end. (See Just in Case You Wanted to Know Box 1.1 for information on disasters caused by other software faults.)

Whether we are dealing with billing or air defense, much of our software is delivered late, over budget, and with residual faults, and does not meet the client's needs. Software engineering is an attempt to solve these problems. In other words, **software engineering** is a discipline whose aim is the production of fault-free software, delivered on time and within budget, that satisfies the client's needs. Furthermore, the software must be easy to modify when the user's needs change.

The scope of software engineering is extremely broad. Some aspects of software engineering can be categorized as mathematics or computer science; other aspects fall into the areas of economics, management, or psychology. To display the wide-reaching realm of software engineering, we now examine five different aspects.

1.1 Historical Aspects

It is a fact that electric power generators fail, but far less frequently than payroll products. Bridges sometimes collapse but considerably less often than operating systems. In the belief that software design, implementation, and maintenance could be put on the same

In the case of the WWMCCS network, disaster was averted at the last minute. However, the consequences of other software faults have been fatal. For example, between 1985 and 1987, at least two patients died as a consequence of severe overdoses of radiation delivered by the Therac-25 medical linear accelerator [Leveson and Turner, 1993]. The cause was a fault in the control software.

Also, during the 1991 Gulf War, a Scud missile penetrated the Patriot antimissile shield and struck a barracks near Dhahran, Saudi Arabia. In all, 28 Americans were killed and 98 wounded. The software for the Patriot missile contained a cumulative timing fault. The Patriot was designed to operate for only a few hours at a time, after which the clock was reset. As a result, the fault never had a significant effect and therefore was not detected. In the Gulf War, however, the Patriot missile battery at Dhahran ran continuously for over 100 hours. This caused the accumulated time discrepancy to become large enough to render the system inaccurate.

During the Gulf War, the United States shipped Patriot missiles to Israel for protection against the Scuds. Israeli forces detected the timing problem after only 8 hours and immediately reported it to the manufacturer in the United States. The manufacturer corrected the fault as quickly as it could, but tragically, the new software arrived the day after the direct hit by the Scud [Mellor, 1994].

Fortunately, it is extremely rare for death or serious injury to be caused by a software fault. However, one fault can cause major problems for thousands and thousands of people. For example, in February 2003, a software fault resulted in the U.S. Treasury Department mailing 50,000 Social Security checks that had been printed without the name of the beneficiary, so the checks could not be deposited or cashed [St. Petersburg Times Online, 2003]. In April 2003, borrowers were informed by SLM Corp. (commonly known as Sallie Mae) that the interest on their student loans had been miscalculated as a consequence of a software fault from 1992 but detected only at the end of 2002. Nearly 1 million borrowers were told that they would have to pay more, either in the form of higher monthly payments or extra interest payments on loans extending beyond their original 10-year terms [GJSentinel.com, 2003]. Both faults were quickly corrected, but together they resulted in nontrivial financial consequences for about a million people.

The Belgian government overestimated its 2007 budget by €883,000,000 (more than $1,100,000,000 at time of writing). This mistake was caused by a software fault compounded by the manual overriding of an error-detection mechanism [La Libre Online, 2007a; 2007b]. The Belgian tax authorities used scanners and optical character recognition software to process tax returns. If the software encountered an unreadable return, it recorded the taxpayer's income as €99,999,999.99 (over $125,000,000). Presumably, the "magic number" €99,999,999.99 was chosen to be quickly detected by employees of the data processing department, so that the return in question would then be processed manually. This worked fine when the tax returns were analyzed for tax assessment purposes, but not when the tax returns were reanalyzed for budgetary purposes. Ironically, the software product did have filters to detect this sort of problem, but the filters were manually bypassed to speed up processing.

There were at least two faults in the software. First, the software engineers assumed that there would always be adequate manual scrutiny before further processing of the data. Second, the software allowed the filters to be manually overridden.

Just in Case You Wanted to Know Box 1.2

As stated in Section 1.1, the aim of the Garmisch conference was to make software development as successful as traditional engineering. But by no means are all traditional engineering projects successful. For example, consider bridge building.

In July 1940, construction of a suspension bridge over the Tacoma Narrows, in Washington State, was completed. Soon after, it was discovered that the bridge swayed and buckled dangerously in windy conditions. Approaching cars would alternately disappear into valleys and then reappear as that part of the bridge rose again. From this behavior, the bridge was given the nickname "Galloping Gertie." Finally, on November 7, 1940, the bridge collapsed in a 42 mile per hour wind; fortunately, the bridge had been closed to all traffic some hours earlier. The last 15 minutes of its life were captured on film, now stored in the U.S. National Film Registry.

A somewhat more humorous bridge construction failure was observed in January 2004. A new bridge was being built over the Upper Rhine River near the German town of Laufenberg, to connect Germany and Switzerland. The German half of the bridge was designed and constructed by a team of German engineers; the Swiss half by a Swiss team. When the two parts were connected, it immediately became apparent that the German half was some 21 inches (54 centimeters) higher than the Swiss half. Major reconstruction was needed to correct the problem, which was caused by wrongly correcting for the fact that "sea level" is taken by Swiss engineers to be the average level of the Mediterranean Sea, whereas German engineers use the North Sea. To compensate for the difference in sea levels, the Swiss side should have been raised 10.5 inches. Instead, it was lowered 10.5 inches, resulting in the gap of 21 inches [Spiegel Online, 2004].

footing as traditional engineering disciplines, a NATO study group in 1967 coined the term *software engineering*. The claim that building software is similar to other engineering tasks was endorsed by the 1968 NATO Software Engineering Conference held in Garmisch, Germany [Naur, Randell, and Buxton, 1976]. This endorsement is not too surprising; the very name of the conference reflected the belief that software production should be an engineering-like activity (but see Just in Case You Wanted to Know Box 1.2). A conclusion of the conferees was that software engineering should use the philosophies and paradigms of established engineering disciplines to solve what they termed the **software crisis**, namely, that the quality of software generally was unacceptably low and that deadlines and budgets were not being met.

Despite many software success stories, an unacceptably large proportion of software products still are being delivered late, over budget, and with residual faults. For example, the Standish Group is a research firm that analyzes software development projects. Their study of development projects completed in 2006 is summarized in Figure 1.1 [Rubenstein, 2007]. Only 35 percent of the projects were successfully completed, whereas 19 percent were canceled before completion or were never implemented. The remaining 46 percent of the projects were completed and installed on the client's computer. However, those projects were over budget, late, or had fewer features and functionality than initially specified. In other words, during 2006, just over one in three software development projects was successful; almost half the projects displayed one or more symptoms of the software crisis.

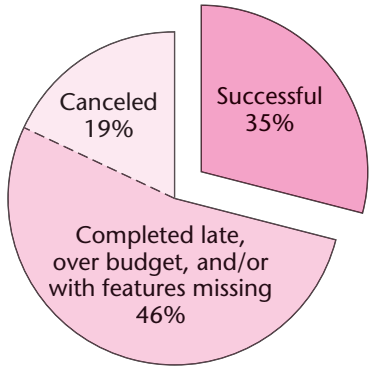


FIGURE 1.1
The outcomes of over 9,000 development projects completed in 2006 [Rubenstein, 2007].

The financial implications of the software crisis are horrendous. In a survey conducted by the Cutter Consortium [2002], the following was reported:

- An astounding 78 percent of information technology organizations have been involved in disputes that ended in litigation.
- In 67 percent of those cases, the functionality or performance of the software products as delivered did not measure up to the claims of the software developers.
- In 56 percent of those cases, the promised delivery date slipped several times.
- In 45 percent of those cases, the faults were so severe that the software product was unusable.

It is clear that far too little software is delivered on time, within budget, fault free, and meeting its client's needs. To achieve these goals, a software engineer has to acquire a broad range of skills, both technical and managerial. These skills have to be applied not just to programming but to every step of software production, from requirements to postdelivery maintenance.

That the software crisis still is with us, some 40 years later, tells us two things. First, the software **process**, that is, the way we produce software, has its own unique properties and problems, even though it resembles traditional engineering in many respects. Second, the software crisis perhaps should be renamed the **software depression**, in view of its long duration and poor prognosis.

We now consider economic aspects of software engineering.

1.2 Economic Aspects

A software organization currently using coding technique CT_{old} discovers that new coding technique CT_{new} would result in code being produced in only nine-tenths of the time needed by CT_{old} and, hence, at nine-tenths the cost. Common sense seems to dictate that CT_{new} is the appropriate technique to use. In fact, although common sense certainly dictates that

the faster technique is the technique of choice, the economics of software engineering may imply the opposite.

- One reason is the cost of introducing new technology into an organization. The fact that coding is 10 percent faster when technique CT_{new} is used may be less important than the costs incurred in introducing CT_{new} into the organization. It may be necessary to complete two or three projects before recouping the cost of training. Also, while attending courses on CT_{new}, software personnel are unable to do productive work. Even when they return, a steep learning curve may be involved; it may take many months of practice with CT_{new} before software professionals become as proficient with CT_{new} as they currently are with CT_{old}. Therefore, initial projects using CT_{new} may take far longer to complete than if the organization had continued to use CT_{old}. All these costs need to be taken into account when deciding whether to change to CT_{new}.
- A second reason why the economics of software engineering may dictate that CT_{old} be retained is the maintenance consequence. Coding technique CT_{new} indeed may be 10 percent faster than CT_{old}, and the resulting code may be of comparable quality from the viewpoint of satisfying the client's current needs. But the use of technique CT_{new} may result in code that is difficult to maintain, making the cost of CT_{new} higher over the life of the product. Of course, if the software developer is not responsible for any postdelivery maintenance, then, from the viewpoint of just that developer, CT_{new} is a more attractive proposition. After all, the use of CT_{new} would cost 10 percent less. The client should insist that technique CT_{old} be used and pay the higher initial costs with the expectation that the total lifetime cost of the software will be lower. Unfortunately, often the sole aim of both the client and the software provider is to produce code as quickly as possible. The long-term effects of using a particular technique generally are ignored in the interests of short-term gain. Applying economic principles to software engineering requires the client to choose techniques that reduce long-term costs.

This example deals with coding, which constitutes less than 10 percent of the software development effort. The economic principles, however, apply to all other aspects of software production as well.

We now consider the importance of maintenance.

1.3 Maintenance Aspects

In this section, we describe maintenance within the context of the software life cycle. A **life-cycle model** is a description of the steps that should be performed when building a software product. Many different life-cycle models have been proposed; several of them are described in Chapter 2. Because it is almost always easier to perform a sequence of smaller tasks than one large task, the overall life-cycle model is broken into a series of smaller steps, called **phases**. The number of phases varies from model to model—from as few as four to as many as eight. In contrast to a life-cycle model, which is a theoretical description of what should be done, the actual series of steps performed on a specific software product, from concept exploration through final retirement, is termed the **life cycle** of that product. In practice, the phases of the life cycle of a software product may not be carried out exactly as specified in the life-cycle model, especially when time and cost overruns

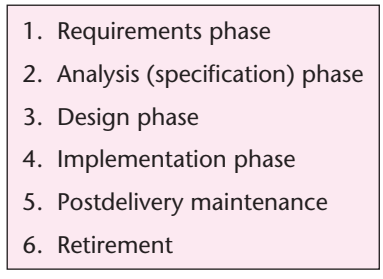


FIGURE 1.2
The six phases of the classical life-cycle model.

are encountered. It has been claimed that more software projects have gone wrong for lack of time than for all other reasons combined [Brooks, 1975].

Until the end of the 1970s, most organizations were producing software using as their life-cycle model what now is termed the **waterfall model**. There are many variations of this model, but by and large, a product developed using this classical life-cycle model goes through the six phases shown in Figure 1.2. These phases probably do not correspond exactly to the phases of any one particular organization, but they are sufficiently close to most practices for the purposes of this book. Similarly, the precise name of each phase varies from organization to organization. The names used here for the various phases have been chosen to be as general as possible in the hope that the reader will feel comfortable with them.

1. *Requirements phase.* During the **requirements phase**, the concept is explored and refined, and the client's requirements are elicited.
2. *Analysis (specification) phase.* The client's requirements are analyzed and presented in the form of the **specification document**, "what the product is supposed to do." The **analysis phase** sometimes is called the **specification phase**. At the end of this phase, a plan is drawn up, the **software project management plan**, describing the proposed software development in full detail.
3. *Design phase*. The specifications undergo two consecutive design procedures during the **design phase**. First comes **architectural design**, in which the product as a whole is broken down into components, called **modules**. Then, each module is designed; this procedure is termed **detailed design**. The two resulting **design documents** describe "how the product does it."
4. *Implementation phase*. The various components undergo **coding** and testing (**unit testing**) separately. Then, the components of the product are combined and tested as a whole; this is termed **integration**. When the developers are satisfied that the product functions correctly, it is tested by the client (**acceptance testing**). The **implementation phase** ends when the product is accepted by the client and installed on the client's computer. (We see in Chapter 15 that coding and integration should be performed in parallel.)
5. *Postdelivery maintenance.* The product is used to perform the tasks for which it was developed. During this time, it is maintained. **Postdelivery maintenance** includes all changes to the product once the product has been delivered and installed on the client's computer and passes its acceptance test. Postdelivery maintenance

Just in Case You Wanted to Know — Box 1.3

One of the most widely quoted results in software engineering is that 17.4 percent of the postdelivery maintenance effort is corrective in nature; 18.2 percent is adaptive; 60.3 percent is perfective; and 4.1 percent can be categorized as "other." This result is taken from a paper published in 1978 [Lientz, Swanson, and Tompkins, 1978].

However, the result in that paper was not derived from *measurements* on maintenance data. Instead, the authors conducted a survey of maintenance managers who were asked to *estimate* how much time was devoted to each category within their organization as a whole and to state how confident they felt about their estimate. More specifically, the participating software maintenance managers were asked whether their response was based on reasonably accurate data, minimal data, or no data; 49.3 percent stated that their answer was based on reasonably accurate data, 37.7 percent on minimal data, and 8.7 percent on no data.

In fact, one should seriously question whether any respondents had "reasonably accurate data" regarding the percentage of time devoted to the categories of maintenance included in the survey; most of them probably did not have even "minimal data." In that survey, participants were asked to state what percentage of maintenance consisted of items like "emergency fixes" or "routine debugging"; from this raw information, the percentage of adaptive, corrective, and perfective maintenance was deduced. Software engineering was just starting to emerge as a discipline in 1978, and it was the exception for software maintenance managers to collect the detailed information needed to respond to such a survey. Indeed, in modern terminology, in 1978 virtually every organization was still at CMM level 1 (see Section 3.13).

Hence, we have strong grounds for questioning whether the actual distribution of postdelivery maintenance activities back in 1978 was anything like the estimates of the managers who took part in the survey. The distribution of maintenance activities is certainly nothing like that today. For example, results on actual maintenance data for the Linux kernel [Schach et al., 2002] and the gcc compiler [Schach et al., 2003] show that at least 50 percent of postdelivery maintenance is corrective, as opposed to the 17.4 percent figure claimed in the survey.

includes **corrective maintenance** (or **software repair**), which consists of the removal of residual faults while leaving the specifications unchanged, as well as **enhancement** (or software update), which consists of changes to the specifications and the implementation of those changes. There are, in turn, two types of enhancement. The first is **perfective maintenance**, changes that the client thinks will improve the effectiveness of the product, such as additional functionality or decreased response time. The second is **adaptive maintenance**, changes made in response to changes in the environment in which the product operates, such as a new hardware/operating system or new government regulations. (For an insight into the three types of postdelivery maintenance, see Just in Case You Wanted to Know Box 1.3.)

6. *Retirement*. **Retirement** occurs when the product is removed from service. This occurs when the functionality provided by the product no longer is of any use to the client organization.

Now we examine the definition of *maintenance* in greater detail.

1.3.1 Classical and Modern Views of Maintenance

In the 1970s, software production was viewed as consisting of two distinct activities performed sequentially: *development* followed by *maintenance*. Starting from scratch, the software product was developed, and then installed on the client's computer. Any change to the software after installation on the client's computer and acceptance by the client, whether to fix a residual fault or extend the functionality, constituted classical maintenance [IEEE 610.12, 1990]. Hence, the way that software was developed classically can be described as the **development-then-maintenance model**.

This is a **temporal definition**; that is, an activity is classified as development or maintenance depending on when it is performed. Suppose that a fault in the software is detected and corrected a day after the software has been installed. By definition, this constitutes classical maintenance. But if the identical fault is detected and corrected the day before the software is installed, in terms of the definition, this constitutes classical development. Now suppose that a software product has just been installed but the client wants to increase the functionality of the software product. Classically, that would be described as perfective maintenance. However, if the client wants the same change to be made just before the software product is installed, this would be classical development. Again, there is no difference whatsoever between the nature of the two activities, but classically one is considered development, the other perfective maintenance.

In addition to such inconsistencies, two other reasons explain why the development-then-maintenance model is unrealistic today:

1. Nowadays, it is certainly not unusual for construction of a product to take a year or more. During this time, the client's requirements may well change. For example, the client might insist that the product now be implemented on a faster processor, which has just become available. Alternatively, the client organization may have expanded into Belgium while development was under way, and the product now has to be modified so it can also handle sales in Belgium. To see how a change in requirements can affect the software life cycle, suppose that the client's requirements change while the design is being developed. The software engineering team has to suspend development and modify the specification document to reflect the changed requirements. Furthermore, it then may be necessary to modify the design as well, if the changes to the specifications necessitate corresponding changes to those portions of the design already completed. Only when these changes have been made can development proceed. In other words, developers have to perform "maintenance" long before the product is installed.
2. A second problem with the classical development-then-maintenance model arose as a result of the way in which we now construct software. In classical software engineering, a characteristic of development was that the development team built the target product starting from scratch. In contrast, as a consequence of the high cost of software production today, wherever possible developers try to reuse parts of existing software products in the software product to be constructed (reuse is discussed in detail in Chapter 8). Therefore, the development-then-maintenance model is inappropriate today because reuse is so widespread.

A more realistic way of looking at maintenance is that given in the standard for life-cycle processes published by the International Organization for Standardization (ISO)

Just in Case You Wanted to Know — Box 1.4

The International Organization for Standardization (ISO) is a network of the national standards institutes of 147 countries, with a central secretariat based in Geneva, Switzerland. ISO has published over 13,500 internationally accepted standards, ranging from standards for photographic film speed ("ISO number") to many of the standards presented in this book. For example, ISO 9000 is discussed in Chapter 3.

ISO is not an acronym. It is derived from the Greek word ἴσος, meaning *equal*, the root of the English prefix *iso-* found in words such as *isotope*, *isobar*, and *isosceles*. The International Organization for Standardization chose ISO as the short form of its name to avoid having multiple acronyms arising from the translation of the name "International Organization for Standardization" into the languages of the different member countries. Instead, to achieve international standardization, a universal short form of its name was chosen.

and the International Electrotechnical Commission (IEC). That is, maintenance is the process that occurs when "software undergoes modifications to code and associated documentation due to a problem or the need for improvement or adaptation" [ISO/IEC 12207, 1995]. In terms of this **operational definition**, maintenance occurs whenever a fault is fixed or the requirements change, irrespective of whether this takes place before or after installation of the product. The Institute for Electrical and Electronics Engineers (IEEE) and the Electronic Industries Alliance (EIA) subsequently adopted this definition [IEEE/EIA 12207.0-1996, 1998] when IEEE standards were modified to comply with ISO/IEC 12207. (See Just in Case You Wanted to Know Box 1.4 for more on ISO.)

In this book, the term *postdelivery maintenance* refers to the 1990 IEEE definition of maintenance as any change to the software after it has been delivered and installed on the client's computer, and *modern maintenance* or just **maintenance** refers to the 1995 ISO/IEC definition of corrective, perfective, or adaptive activities performed at any time. Postdelivery maintenance is therefore a subset of (modern) maintenance.

1.3.2 The Importance of Postdelivery Maintenance

It is sometimes said that only bad software products undergo postdelivery maintenance. In fact, the opposite is true: Bad products are thrown away, whereas good products are repaired and enhanced, for 10, 15, or even 20 years. Furthermore, a software product is a model of the real world, and the real world is perpetually changing. As a consequence, software has to be maintained constantly for it to remain an accurate reflection of the real world.

For instance, if the sales tax rate changes from 6 to 7 percent, almost every software product that deals with buying or selling has to be changed. Suppose the product contains the C++ statement

```
const float salesTax = 6.0;
```

or the equivalent Java statement

```
public static final float salesTax = (float) 6.0;
```

declaring that salesTax is a floating-point constant initialized to the value 6.0. In this case, maintenance is relatively simple. With the aid of a text editor the value 6.0 is replaced by 7.0 and the code is recompiled and relinked. However, if instead of using the name salesTax, the actual value 6.0 has been used in the product wherever the value of the sales tax is invoked, then such a product is extremely difficult to modify. For example, there may be occurrences of the value 6.0 in the source code that should be changed to 7.0 but are overlooked, or instances of 6.0 that do not refer to sales tax but are incorrectly changed to 7.0. Finding these faults almost always is difficult and time consuming. In fact, with some software, it might be less expensive in the long run to throw away the product and recode it rather than try to determine which of the many constants need to be changed and how to make the modifications.

The real-time real world also is constantly changing. The missiles with which a jet fighter is armed may be replaced by a new model, requiring a change to the weapons control component of the associated avionics system. A six-cylinder engine is to be offered as an option in a popular four-cylinder automobile; this implies changing the onboard computers that control the fuel injection system, timing, and so on.

But just how much time (= money) is devoted to postdelivery maintenance? The pie chart in Figure 1.3(a) shows that, some 40 years ago, approximately two-thirds of total software costs went to postdelivery maintenance; the data were obtained by averaging information from various sources, including [Elshoff, 1976], [Daly, 1977], [Zelkowitz, Shaw, and Gannon, 1979], and [Boehm, 1981]. Newer data show that an even larger proportion is devoted to postdelivery maintenance. Many organizations devote 70–80 percent or more of their software budget to postdelivery maintenance [Yourdon, 1992; Hatton, 1998], as shown in Figure 1.3(b).

Surprisingly, the average cost percentages of the classical development phases have hardly changed. This is shown in Figure 1.4, which compares the data used to derive Figure 1.3(a) with more recent data on 132 Hewlett-Packard projects [Grady, 1994].

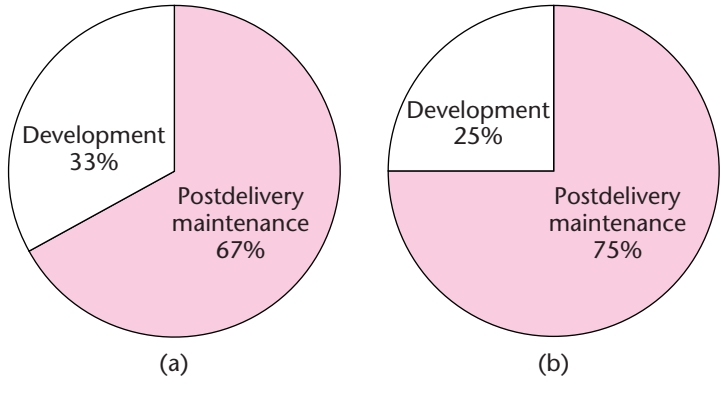


FIGURE 1.3
Approximate average cost percentages of development and postdelivery maintenance (a) between 1976 and 1981 and (b) between 1992 and 1998.

FIGURE 1.4 A comparison of the approximate average cost percentages of the classical development phases for various projects between 1976 and 1981 and for 132 more recent Hewlett-Packard projects.

	Various Projects between 1976 and 1981	132 More Recent Hewlett-Packard Projects
Requirements and analysis (specification) phases	21%	18%
Design phase	18	19
Implementation phase		
Coding (including unit testing)	36	34
Integration	24	29

Now consider again the software organization currently using coding technique CT_{old} that learns that CT_{new} will reduce coding time by 10 percent. Even if CT_{new} has no adverse effect on maintenance, an astute software manager will think twice before changing coding practices. The entire staff has to be retrained, new software development tools purchased, and perhaps additional staff members hired who are experienced in the new technique. All this expense and disruption has to be endured for a decrease of at most 0.85 percent in software costs because, as shown in Figures 1.3(b) and 1.4, coding together with unit testing constitutes on average only 34 percent of 25 percent or 8.5 percent of total software costs.

Now suppose a new technique that reduces postdelivery maintenance costs by 10 percent is developed. This probably should be introduced at once, because on average, it will reduce overall costs by 7.5 percent. The overhead involved in changing to this technique is a small price to pay for such large overall savings.

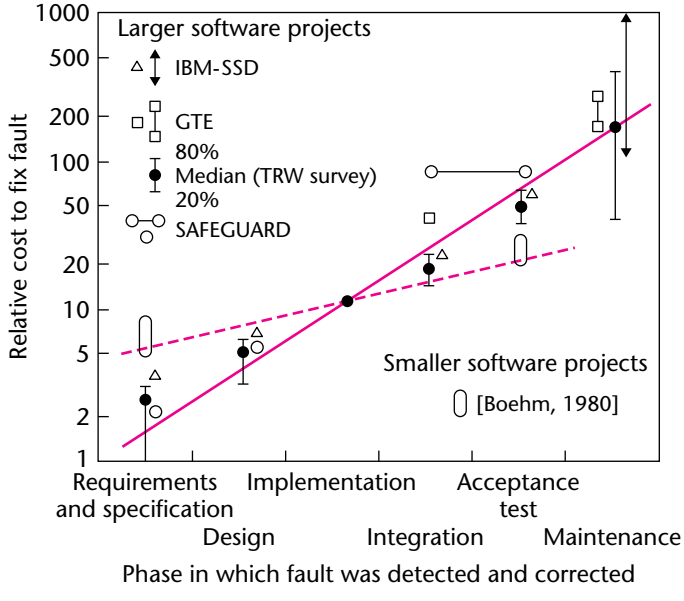
Because postdelivery maintenance is so important, a major aspect of software engineering consists of those techniques, tools, and practices that lead to a reduction in postdelivery maintenance costs.

1.4 Requirements, Analysis, and Design Aspects

Software professionals are human and therefore sometimes make a mistake while developing a product. As a result, there will be a fault in the software. If the mistake is made while eliciting the requirements, the resulting fault will probably also appear in the specifications, the design, and the code. Clearly, the earlier we correct a fault, the better.

The relative costs of fixing a fault at various phases in the classical software life cycle are shown in Figure 1.5 [Boehm, 1981]. The figure reflects data from IBM [Fagan, 1974], GTE [Daly, 1977], the Safeguard project [Stephenson, 1976], and some smaller TRW projects [Boehm, 1980]. The solid line in Figure 1.5 is the best fit for the data relating to the larger projects, and the dashed line is the best fit for the smaller projects. For each of the phases of the classical software life cycle, the corresponding relative cost to detect and correct a

FIGURE 1.5 The relative cost of fixing a fault at each phase of the classical software life cycle. The solid line is the best fit for the data relating to the larger software projects, and the dashed line is the best fit for the smaller software projects. (Barry Boehm, *Software Engineering Economics,* © 1981, p. 40. Adapted by permission of Prentice Hall, Inc., Englewood Cliffs, NJ.)



fault is depicted in Figure 1.6. Each step on the solid line in Figure 1.6 is constructed by taking the corresponding point on the solid straight line of Figure 1.5 and plotting the data on a linear scale.

Suppose it costs $40 to detect and correct a specific fault during the design phase. From the solid line in Figure 1.6 (projects between 1974 and 1980), that same fault would cost only about $30 to fix during the analysis phase. But during postdelivery maintenance, that fault would cost around $2000 to detect and correct. Newer data show that now it is even more important to detect faults early. The dashed line in Figure 1.6 shows the cost of detecting and correcting a fault during the development of system software for the IBM AS/400 [Kan et al., 1994]. On average, the same fault would have cost $3680 to fix during postdelivery maintenance of the AS/400 software.

The reason that the cost of correcting a fault increases so steeply is related to what has to be done to correct a fault. Early in the development life cycle, the product essentially exists only on paper, and correcting a fault may simply mean making a change to a document. The other extreme is a product already delivered to a client. At the very least, correcting a fault at that time means editing the code, recompiling and relinking it, and then carefully testing that the problem is solved. Next, it is critical to check that making the change has not created a new problem elsewhere in the product. All the relevant documentation, including manuals, needs to be updated. Finally, the corrected product must be delivered

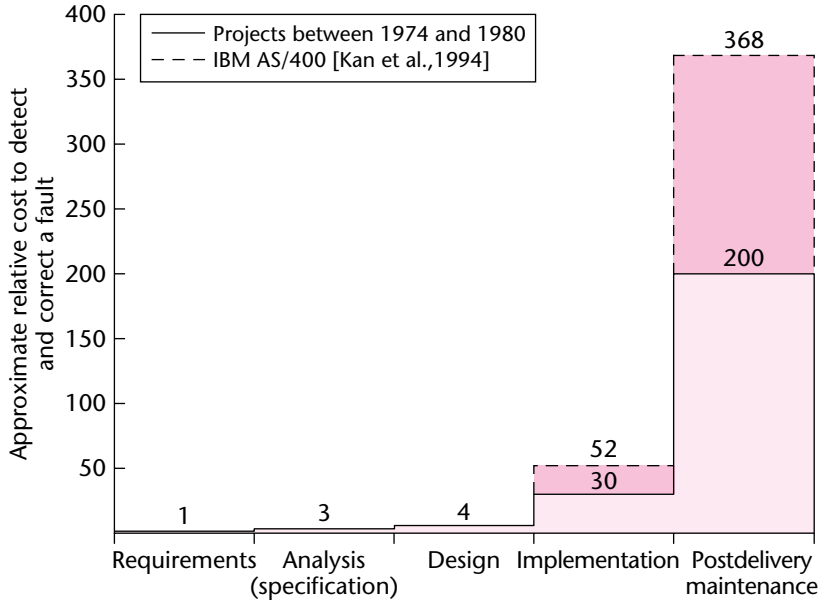


FIGURE 1.6
The solid line depicts the points on the solid line of Figure 1.5 plotted on a linear scale. The dashed line depicts newer data.

and reinstalled. The moral of the story is this: We must find faults early or else it will cost us money. We therefore should employ techniques for detecting faults during the requirements and analysis (specification) phases.

There is a further need for such techniques. Studies have shown [Boehm, 1979] that between 60 and 70 percent of all faults detected in large projects are requirements, analysis, or design faults. Newer results from inspections bear out this preponderance of requirements, analysis, or design faults (an inspection is a meticulous examination of a document by a team, as described in Section 6.2.3). During 203 inspections of Jet Propulsion Laboratory software for the NASA unmanned interplanetary space program, on average, about 1.9 faults were detected per page of a specification document, 0.9 faults per page of a design, but only 0.3 faults per page of code [Kelly, Sherif, and Hops, 1992].

Therefore it is important that we improve our requirements, analysis, and design techniques, not only so that faults can be found as early as possible but also because requirements, analysis, and design faults constitute such a large proportion of all faults. Just as the example in Section 1.3 showed that reducing postdelivery maintenance costs by 10 percent reduces overall costs by about 7.5 percent, reducing requirements, analysis, and design faults by 10 percent reduces the overall number of faults by 6–7 percent.

That so many faults are introduced early in the software life cycle highlights another important aspect of software engineering: techniques that yield better requirements, specifications, and designs.

Most software is produced by a team of software engineers rather than by a single individual responsible for every aspect of the development and maintenance life cycle. We now consider the implications of this.

1.5 Team Development Aspects

The cost of hardware continues to decrease rapidly. A mainframe computer of the 1950s that cost in excess of a million preinflation dollars was considerably less powerful in every way than a laptop computer of today costing less than $1000. As a result, organizations easily can afford hardware that can run large products, that is, products too large (or too complex) to be implemented by one person within the allowed time constraints. For example, if a product has to be delivered within 18 months but would take a single software professional 15 years to complete, then the product must be developed by a team. However, team development leads to interfacing problems among code components and communication problems among team members.

For example, Jeff and Juliet code modules p and q, respectively, where module p calls module q. When Jeff codes p, he inserts a call to q with five arguments in the argument list. Juliet codes q with five arguments, but in a different order from those of Jeff. Some software tools, such as the Java interpreter and loader, or *lint* for C (Section 8.11.4), detect such a type violation but only if the interchanged arguments are of different types; if they are of the same type, then the problem may not be detected for a long period of time. It may be debated that this is a design problem, and if the modules had been more carefully designed, this problem would not have happened. That may be true, but in practice a design often is changed after coding commences, and notification of a change may not be distributed to all members of the development team. Therefore, when a design that affects two or more programmers has been changed, poor communication can lead to the interface problems Jeff and Juliet experienced. This sort of problem is less likely to occur when only one individual is responsible for every aspect of the product, as was the case before powerful computers that can run huge products became affordable.

But interfacing problems are merely the tip of the iceberg when it comes to problems that can arise when software is developed by teams. Unless the team is properly organized, an inordinate amount of time can be wasted in conferences between team members. Suppose that a product takes a single programmer 1 year to complete. If the same task is assigned to a team of six programmers, the time for completing the task frequently is closer to 1 year than the expected 2 months, and the quality of the resulting code may well be lower than if the entire task had been assigned to one individual (see Section 4.1). Because a considerable proportion of today's software is developed and maintained by teams, the scope of software engineering must include techniques for ensuring that teams are properly organized and managed.

As has been shown in the preceding sections, the scope of software engineering is extremely broad. It includes every step of the software life cycle, from requirements to postdelivery retirement. It also includes human aspects, such as team organization; economic aspects; and legal aspects, such as copyright law. All these aspects implicitly are incorporated in the definition of software engineering given at the beginning of this chapter, that software engineering is a discipline whose aim is the production of fault-free software delivered on time, within budget, and satisfying the user's needs.

We return to the classical phases of Figure 1.2 to ask why there is no planning, testing, or documentation phase.

1.6 Why There Is No Planning Phase

Clearly it is impossible to develop a software product without a plan. Accordingly, it appears to be essential to have a **planning phase** at the very beginning of the project.

The key point is that, until it is known exactly what is to be developed, there is no way an accurate, detailed plan can be drawn up. Therefore, three types of planning activities take place when a software product is developed using the classical paradigm:

1. At the beginning of the project, preliminary planning takes place for managing the requirements and analysis phases.
2. Once what is going to be developed is known precisely, the *software project management plan* (SPMP) is drawn up. This includes the budget, staffing requirements, and detailed schedule. The earliest we can draw up the project management plan is when the specification document has been approved by the client, that is, at the end of the analysis phase. Until that time, planning has to be preliminary and partial.
3. All through the project, management needs to monitor the SPMP and be on the watch for any deviation from the plan.

For example, suppose that the SPMP for a specific project states that the project as a whole will take 16 months and that the design phase will take 4 of those months. After a year, management notices that the project as a whole seems to be progressing much more slowly than anticipated. A detailed investigation shows that, so far, 8 months have been devoted to the design phase, which is still far from complete. The project almost certainly will have to be abandoned, and the funds spent to date are wasted. Instead, management should have tracked progress by phase, and noticed, after at most 2 months, a serious problem in the design phase. At that time, a decision could have been made how best to proceed. The usual initial step in such a situation is to call in a consultant to determine if the project is feasible and to determine whether the design team is competent to carry out the task or the risk of proceeding is too great. Based on the report of the consultant, various alternatives are now considered, including reducing the scope of the target product, and then designing and implementing a less ambitious one. Only if all other alternatives are considered unworkable does the project have to be canceled. In the case of the specific project, this cancellation would have taken place some 6 months earlier if management had monitored the plan closely, saving a considerable sum of money.

In conclusion, there is no separate planning phase. Instead, planning activities are carried out all through the life cycle. However, there are times when planning activities predominate. These include the beginning of the project (preliminary planning) and directly after the specification document has been signed off on by the client (software project management plan).

1.7 Why There Is No Testing Phase

It is essential to check a software product meticulously after it has been developed. Accordingly, it is reasonable to ask why there is no testing phase after the product has been implemented.

Unfortunately, checking a software product once it is ready to be delivered to the client is far too late. For instance, if there is a fault in the specification document, this fault will have been carried forward into the design and implementation. There are times in the software process when testing is carried out almost to the total exclusion of other activities. This occurs toward the end of each phase (**verification**) and is especially true before the product is handed over to the client (**validation**). Although there are times when testing predominates, there should never be times when no testing is being performed. If testing is treated as a separate (**testing**) **phase**, then there is a very real danger that testing will not be carried out constantly throughout every phase of the product development and maintenance process.

But even this is not enough. What is needed is continual checking of a software product. Meticulous checking should automatically accompany every software development and maintenance activity. A separate testing phase is incompatible with the goal of ensuring that a software product is as fault free as possible at all times.

Every software development organization should contain an independent group whose primary responsibility is to ensure that the delivered product is what the client needs and that the product has been built correctly in every way. This group is called the *software quality assurance* (SQA) group. The **quality** of software is the extent to which it meets its specifications. Quality and software quality assurance are described in more detail in Chapter 6, as is the role of SQA in setting and enforcing standards.

1.8 Why There Is No Documentation Phase

Just as there should never be a separate planning phase or testing phase, there also should never be a separate **documentation phase**. On the contrary, at all times, the documentation of a software product must be complete, correct, and up to date. For instance, during the analysis phase, the specification document must reflect the current version of the specifications, and this is also true for the other phases.

1. One reason why it is essential to ensure that the documentation is always up to date is the large turnover in personnel in the software industry. For example, suppose that the design documentation has not been kept current and the chief designer leaves to take another job. It is now extremely hard to update the design document to reflect all the changes made while the system was being designed.
2. It is almost impossible to perform the steps of a specific phase unless the documentation of the previous phase is complete, correct, and up to date. For instance, an incomplete specification document must inevitably result in an incomplete design and then in an incomplete implementation.
3. It is virtually impossible to test whether a software product is working correctly unless documents are available that state how that software product is supposed to behave.
4. Maintenance is almost impossible unless there is a complete and correct set of documentation that describes precisely what the current version of the product does.

Therefore, just as there is no separate planning phase or testing phase, there is no separate documentation phase. Instead, planning, testing, and documentation should be activities that accompany all other activities while a software product is being constructed.

Now we examine the object-oriented paradigm.

1.9 The Object-Oriented Paradigm

Before 1975, most software organizations used no specific techniques; each individual worked his or her own way. Major breakthroughs were made between approximately 1975 and 1985, with the development of the so-called **structured** or **classical paradigm**. The techniques constituting the classical paradigm include structured systems analysis (Section 12.3), data flow analysis (Section 14.3), structured programming, and structured testing (Section 15.13.2). These techniques seemed extremely promising when first used. However, as time passed, they proved to be somewhat less successful in two respects:

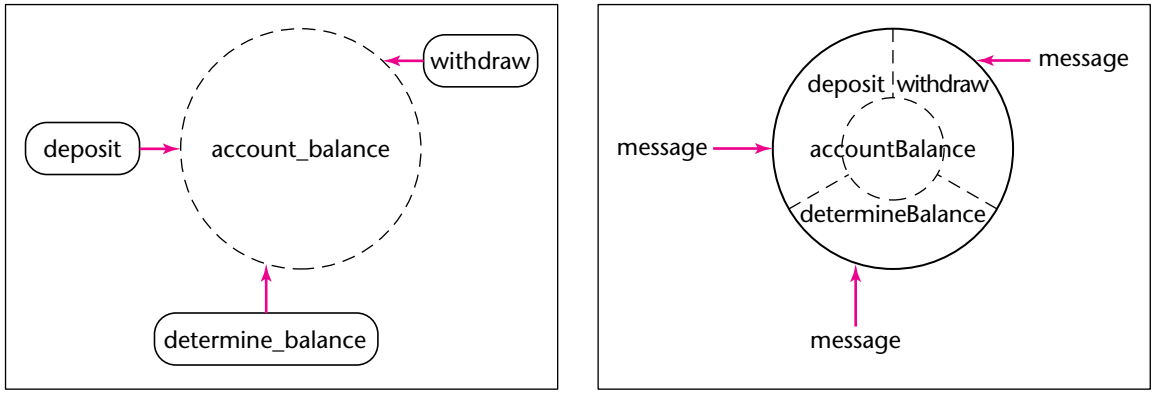
1. The techniques sometimes were unable to cope with the increasing size of software products. That is, the classical techniques were adequate when dealing with small-scale products (typically 5000 lines of code) or even medium-scale products of 50,000 lines of code. Today, however, large-scale products of 500,000 lines of code are relatively common; even products of 5 million or more lines of code are not considered unusual. However, the classical techniques frequently could not scale up sufficiently to handle the development of today's larger products.
2. The classical paradigm did not live up to earlier expectations during postdelivery maintenance. A major driving force behind the development of the classical paradigm some 40 years ago was that, on average, two-thirds of the software budget was being devoted to postdelivery maintenance (see Figure 1.3). Unfortunately, the classical paradigm has not solved this problem; as pointed out in Section 1.3.2, many organizations still spend 70–80 percent or more of their time and effort on postdelivery maintenance [Yourdon, 1992; Hatton, 1998].

A major reason for the limited success of the classical paradigm is that classical techniques are either operation oriented or attribute (data) oriented but not both. The basic components of a software product are the operations of the product and the attributes on which those operations operate. For example, determine_average_height[1] is an operation that operates on a collection of heights (attributes) and returns the average of those heights (attribute). Some classical techniques, such as data flow analysis (Section 14.3), are operation oriented. That is, such techniques concentrate on the operations of the product; the attributes are of secondary importance. Conversely, techniques such as Jackson system development (Section 14.5) are attribute oriented. The emphasis here is on the attributes; the operations that operate on the attributes are less significant.

In contrast, the object-oriented paradigm considers both attributes and operations to be equally important. A simplistic way of looking at an object is as a unified software artifact that incorporates both the attributes and the operations performed on the attributes (an **artifact** is a component of a software product, such as a specification document, a code module, or a manual). This definition of an object is incomplete and is fleshed out later in the book, once *inheritance* has been defined (Section 7.8). Nevertheless, the definition captures much of the essence of an object.

[1]In this book, the name of a variable in a classical software product is written using the classical convention of separating the parts of a variable name with underscores, for example, this_is_a_classical_variable. A variable in an object-oriented software product is written using the object-oriented convention of using an uppercase letter to mark the start of a new part of the name of a variable; for example, thisIsAnObjectOrientedVariable.

FIGURE 1.7 A comparison of implementations of a bank account using (a) the classical paradigm and (b) the object-oriented paradigm. The solid black line surrounding the object denotes that details as to how accountBalance is implemented are not known outside the object.



A bank account is one example of an object (see Figure 1.7). The attribute component of the object is the accountBalance. The operations that can be performed on that account balance include deposit money in the account, withdraw money from the account, and determineBalance. The bank account object combines an attribute with the three operations performed on that attribute in a single artifact. From the viewpoint of the classical paradigm, a product that deals with banking would have to incorporate an attribute, the account_balance, and three operations, deposit, withdraw, and determine_balance.

Up to now, there seems to be little difference between the two approaches. However, a key point is the way in which an object is implemented. Specifically, details as to how the attributes of an object are stored are not known from outside the object. This is an instance of "information hiding," discussed in more detail in Section 7.6. In the case of the bank account object shown in Figure 1.7(b), the rest of the software product is aware that there is such a thing as a balance within a bank account object, but it has no idea as to the format of accountBalance. That is, there is no knowledge outside the object as to whether the account balance is implemented as an integer or a floating-point number or a field (component) of some larger structure. This information barrier surrounding the object is denoted by the solid black line in Figure 1.7(b), which depicts an implementation using the object-oriented paradigm. In contrast, a dashed line surrounds account_balance in Figure 1.7(a), because all the details of account_balance are known to the modules in the implementation using the classical paradigm, and the value of account_balance therefore can be changed by any of them.

Returning to Figure 1.7(b), the object-oriented implementation, if a customer deposits $10 in an account, then a **message** is sent to the deposit method of the relevant object telling it to increment the accountBalance attribute by $10 (a **method** is an implementation of an operation). The deposit method is within the bank account object and knows how the accountBalance is implemented; this is denoted by the dashed circular line inside the

object. But no entity external to the object needs this knowledge. That the three methods in Figure 1.7(b) shield accountBalance from the rest of the product symbolizes this localization of knowledge. The fact that implementation details are local to an object illustrates the first of the many strengths of the object-oriented paradigm:

1. Consider postdelivery maintenance. Suppose that the banking product has been constructed using the classical paradigm. If the way an account_balance is represented is changed from (say) an integer to a field of a structure, then every part of that product that has anything to do with an account_balance has to be changed, and these changes have to be made consistently. In contrast, if the object-oriented paradigm is used, then changes need be made only within the bank account object itself. No other part of the product has knowledge of how an accountBalance is implemented, so no other part can have access to an accountBalance. Consequently, no other part of the banking product needs to be changed. Accordingly, the object-oriented paradigm makes maintenance quicker and easier, and the chance of introducing a **regression fault** (that is, a fault inadvertently introduced into one part of a product as a consequence of making an apparently unrelated change to another part of the product) is greatly reduced.
2. In addition to maintenance, the object-oriented paradigm also makes development easier. In many instances, an object has a physical counterpart. For example, a bank account object in the bank product corresponds to an actual bank account in the bank for which this product is being implemented. As will be shown in Part B, modeling plays a major role in the object-oriented paradigm. The close correspondence between the objects in a product and their counterparts in the real world should lead to better-quality software.
3. Well-designed objects are independent units. As has been explained, an object consists of both attributes and the operations performed on the attributes. If all the operations performed on the attributes of an object are included in that object, then the object can be considered a conceptually independent entity. Everything in the product that relates to the portion of the real world modeled by that object can be found in the object itself. This conceptual independence sometimes is termed **encapsulation** (Section 7.4). But there is an additional form of independence, physical independence. In a well-designed object, information hiding ensures that implementation details are hidden from everything outside that object. The only allowable form of communication is sending a message to the object to carry out a specific operation. The way that the operation is carried out is entirely the responsibility of the object itself. For this reason, object-oriented design sometimes is referred to as **responsibility-driven design** [Wirfs-Brock, Wilkerson, and Wiener, 1990] or **design by contract** [Meyer, 1992]. (For another view of responsibility-driven design, see Just in Case You Wanted to Know Box 1.5, derived from an example in [Budd, 2002].) Another way of looking at both encapsulation and information hiding is as instances of separation of concerns (Section 5.4).
4. A product built using the classical paradigm is implemented as a set of modules, but conceptually it is essentially a single unit. This is one reason why the classical paradigm has been less successful when applied to larger products. In contrast, when the object-oriented paradigm is used correctly, the resulting product consists of a number of smaller, largely independent units. The object-oriented paradigm reduces the level of complexity of a software product and hence simplifies both development and maintenance.

Just in Case You Wanted to Know Box 1.5

Suppose that you live in New Orleans, and you want to send a Mother's Day bouquet to your mother in Chicago. One strategy would be to consult the Chicago yellow pages (on the World Wide Web), determine which florist is located closest to your mother's apartment, and place your order with that florist. A more convenient way is to order the flowers at **1-800-flowers.com,** leaving the total responsibility for delivering the flowers to that company. It is irrelevant where **1-800-flowers.com** is physically located or which florist is given your order to deliver. In any event, the company does not divulge that information, an instance of information hiding.

In exactly the same way, when a message is sent to an object, not only is it entirely irrelevant how the request is carried out, but the unit that sends the message is not even allowed to know the internal structure of the object. The object itself is entirely responsible for every detail of carrying out the message.

5. The object-oriented paradigm promotes reuse; because objects are independent entities, they can generally be utilized in future products (but see Problem 1.17). This reuse of objects reduces the time and cost of both development and maintenance, as explained in Chapter 8.

When the object-oriented paradigm is utilized, the classical software life cycle of Figure 1.2 has to be modified. Figure 1.8 compares the life-cycle model of the classical paradigm with that of the object-oriented paradigm.

The first difference appears to be purely terminological; the word *phase* is used for the classical paradigm, whereas *workflow* is used for the object-oriented paradigm. In fact, as will be explained in detail in Chapter 2, there is no correspondence between a phase and a workflow. On the contrary, the two terms are totally distinct, and this distinction epitomizes the differences between the life-cycle models that underlie the two paradigms.

In this chapter, we consider another difference between the two paradigms, the role played by modules (in the classical paradigm) versus that played by objects (in the object-oriented paradigm). First consider the design phase of the classical paradigm. As stated in Section 1.3, this phase is divided into two subphases: architectural design followed by detailed design. In the architectural design subphase, the product is decomposed into components, called *modules*. Then, during the detailed design subphase, the data structures and algorithms of each module are designed in turn. Finally, during the implementation phase, these modules are implemented.

If the object-oriented paradigm is used instead, one of the steps of the object-oriented analysis workflow is to determine the classes. Because a class is a kind of module, architectural design is performed during the object-oriented analysis workflow.

FIGURE 1.8 Comparison of the life-cycle models of the classical paradigm and the object-oriented paradigm.

Classical Paradigm	Object-Oriented Paradigm
1. Requirements phase	1. Requirements workflow
2. Analysis (specification) phase	2′. Object-oriented analysis workflow
3. Design phase	3′. Object-oriented design workflow
4. Implementation phase	4′. Object-oriented implementation workflow
5. Postdelivery maintenance	5. Postdelivery maintenance
6. Retirement	6. Retirement

Classical Paradigm	Object-Oriented Paradigm
2. Analysis (specification) phase • Determine what the product is to do	2′. Object-oriented analysis workflow • Determine what the product is to do • Extract the classes
3. Design phase • Architectural design (extract the modules) • Detailed design	3′. Object-oriented design workflow • Detailed design
4. Implementation phase • Code the modules in an appropriate programming language • Integrate	4′. Object-oriented implementation workflow • Code the classes in an appropriate object-oriented programming language • Integrate

FIGURE 1.9 Differences between the classical paradigm and the object-oriented paradigm.

Consequently, object-oriented analysis goes further than the corresponding analysis (specification) phase of the classical paradigm. This is shown in Figure 1.9.

This difference between the two paradigms has major consequences. When the classical paradigm is used, there almost always is a sharp transition between the analysis phase and the design phase. After all, the aim of the analysis phase is to determine *what* the product is to do, whereas the purpose of the design phase is to decide *how* to do it. In contrast, when object-oriented analysis is used, objects enter the life cycle from the very beginning. The objects are extracted in the analysis workflow, designed in the design workflow, and coded in the implementation workflow. The object-oriented paradigm is therefore an integrated approach; the transition from workflow to workflow is far smoother than with the classical paradigm, reducing the number of faults introduced during development.

As already mentioned, it is inadequate to define an object merely as a software artifact that encapsulates both attributes and operations and implements the principle of information hiding. A more complete definition is given in Chapter 7, where objects are examined in depth.

1.10 The Object-Oriented Paradigm in Perspective

Figure 1.1 is evidence of the many shortcomings of the classical (structured) paradigm. However, the object-oriented paradigm is by no means a panacea for all ills:

- Like all approaches to software production, the object-oriented paradigm has to be used correctly; it is just as easy to misuse the object-oriented paradigm as any other paradigm.
- When correctly applied, the object-oriented paradigm can solve some (but not all) of the problems of the classical paradigm.
- The object-oriented paradigm has some problems of its own, as described in Section 7.9.
- The object-oriented paradigm is the best approach available today. However, like all technologies, it is certain to be superseded by a superior technology in the future.

In this book, strengths and weaknesses of both the classical and the object-oriented paradigm are pointed out within the context of the specific topic under discussion. Consequently, the comparison of the two paradigms does not appear in one single place but is spread over the entire book.

We now define a number of software engineering terms.

1.11 Terminology

The **client** is the individual who wants a product to be built (developed). The **developers** are the members of a team responsible for building that product. The developers may be responsible for every aspect of the software process, from the requirements onward, or they may be responsible for only the implementation of an already designed product.

Both the client and developers may be part of the same organization. For example, the client may be the head actuary of an insurance company and the developers a team headed by the vice-president for software development of that insurance company. This is termed **internal software development**. On the other hand, with **contract software** the client and developers are members of totally independent organizations. For instance, the client may be a senior official in the Department of Defense and the developers employees of a major defense contractor specializing in software for weapons systems. On a much smaller scale, the client may be an accountant in a one-person practice and the developer a student who earns income by developing software on a part-time basis.

The third party involved in software production is the **user**. The user is the person or persons on whose behalf the client has commissioned the product and who will utilize the software. In the insurance company example, the users may be insurance agents, who will use the software to select the most appropriate policies. In some instances, the client and the user are the same person (for example, the accountant discussed previously).

As opposed to expensive custom software developed for one client, multiple copies of software, such as word processors or spreadsheets, are sold at much lower prices to a large numbers of buyers. That is, the manufacturers of such software (such as Microsoft or Borland) recover the cost of developing a product by volume selling. This type of software usually is called **commercial off-the-shelf (COTS) software**. The earlier term for this type of software was **shrink-wrapped software** because the box containing the CD or diskettes, the manuals, and the license agreement almost always was shrink-wrapped. Nowadays, COTS software often is downloaded over the World Wide Web—there is no box to shrink-wrap. For this reason, COTS software nowadays sometimes is referred to as **clickware**. COTS software is developed for "the market"; that is, the software is not targeted to a specific client or users until it has been developed and is available for purchase.

Open-source software is becoming extremely popular. An open-source software product is developed and maintained by a team of volunteers and may be downloaded and used free of charge by anyone. Widely used open-source products include the Linux operating system, the Firefox Web browser, and the Apache Web server. The term *open source* refers to the availability of the source code to all, unlike most commercial products where only the executable version is sold. Because any user of an open-source product can scrutinize the source code and report faults to the developers, many open-source software products are of high quality. The expected consequence of the public nature of faults in open-source software was formalized by Raymond in *The Cathedral and the Bazaar* as *Linus's Law*, named after Linus Torvalds, the creator of Linux [Raymond, 2000]. **Linus's Law** states that "given enough eyeballs, all bugs are shallow." In other words, if enough individuals scrutinize the source code of an open-source software product, someone should be able to locate that fault and suggest how to fix it (but see Just in Case You Wanted to Know Box 1.6). A related principle is "Release early. Release often" [Raymond, 2000].

It is self-evident that the more people who carefully examine a piece of code, the more likely it is that someone will be able to find and fix a fault in that code. Accordingly, Linus's Law should perhaps be called "Torvalds's Truism."

That is, open-source developers tend to spend less time on testing than closed-source developers, preferring to release a new version of a product virtually as soon as it is finished, leaving much of the responsibility for testing to users.

A word used on almost every page of this book is **software**. Software consists of not just code in machine-readable form but also all the documentation that is an intrinsic component of every project. Software includes the specification document, the design document, legal and accounting documents of all kinds, the software project management plan, and other management documents as well as all types of manuals.

Since the 1970s, the difference between a **program** and a **system** has become blurred. In the "good old days," the distinction was clear. A program was an autonomous piece of code, generally in the form of a deck of punched cards that could be executed. A system was a related collection of programs. A system might consist of programs P, Q, R, and S. Magnetic tape T_1 was mounted, and then program P was run. It caused a deck of data cards to be read in and produced as output tapes T_2 and T_3. Tape T_2 then was rewound, and program Q was run, producing tape T_4 as output. Program R now merged tapes T_3 and T_4 into tape T_5; T_5 served as input for program S, which printed a series of reports.

Compare that situation with a product, running on a machine with a front-end communications processor and a back-end database manager, that performs real-time control of a steel mill. The single piece of software controlling the steel mill does far more than the old-fashioned system, but in terms of the classic definitions of program and system, this software undoubtedly is a program. To add to the confusion, the term *system* now is also used to denote the hardware–software combination. For example, the flight control system in an aircraft consists of both the in-flight computers and the software running on them. Depending on who is using the term, the flight control system also may include the controls, such as the joystick, that send commands to the computer and the parts of the aircraft, such as the wing flaps, controlled by the computer. Furthermore, within the context of traditional software development, the term **systems analysis** refers to the first two phases (requirements and analysis phases) and **systems design** refers to the third phase (design phase).

To minimize confusion, this book uses the term **product** to denote a nontrivial piece of software. There are two reasons for this convention. The first is simply to obviate the program versus system confusion by using a third term. The second reason is more important. This book deals with the process of software production, that is, the way we produce software, and the end result of a process is termed a *product*. Finally, the term *system* is used in its modern sense, that is, the combined hardware and software, or as part of universally accepted phrases, such as operating system and management information system.

Two words widely used within the context of software engineering are *methodology* and *paradigm*. In the 1970s, the word **methodology** began to be used in the sense of "a way of developing a software product"; the word actually means the "science of methods." Then, in the 1980s, the word **paradigm** became a major buzzword of the business world, as in the phrase, "It's a whole new paradigm." The software industry soon

Just in Case You Wanted to Know — Box 1.7

The first use of the word *bug* to denote a fault is attributed to the late Rear Admiral Grace Murray Hopper, one of the designers of COBOL. On September 9, 1945, a moth flew into the Mark II computer that Hopper and her colleagues used at Harvard and lodged between the contact plates of a relay. Accordingly, there was actually a bug in the system. Hopper taped the bug to the logbook and wrote, "First actual case of bug being found." The logbook, with moth still attached, is in the Naval Museum at the Naval Surface Weapons Center, in Dahlgren, Virginia.

Although this may have been the first use of *bug* in a computer context, the word was used in engineering slang in the 19th century [Shapiro, 1994]. For example, Thomas Alva Edison wrote on November 18, 1878, "This thing gives out and then that—'Bugs'—as such little faults and difficulties are called . . ." [Josephson, 1992]. One of the definitions of *bug* in the 1934 edition of *Webster's New English Dictionary* is, "A defect in apparatus or its operation." It is clear from Hopper's remark that she, too, was familiar with the use of the word in that context; otherwise, she would have explained what she meant.

started using the word *paradigm* in the phrases **object-oriented paradigm** and classical (or **traditional**) **paradigm** to mean "a style of software development." This was another unfortunate choice of terminology, because a paradigm is a model or a pattern. Erudite readers offended by this corruption of the English language are warmly invited to take up the cudgels of linguistic accuracy on the author's behalf; he is tired of tilting at windmills.

A methodology or a paradigm is a component of the software process as a whole. In contrast, a **technique** is a component of a portion of the software process. Examples include coding techniques, documentation techniques, and planning techniques.

When a programmer makes a **mistake**, the consequence of that mistake is a **fault** in the code. Executing the software product then results in a **failure**, that is, the observed incorrect behavior of the product as a consequence of the fault. An **error** is the amount by which a result is incorrect. The terms *mistake*, *fault*, *failure*, and *error* are defined in IEEE Standard 610.12, "A Glossary of Software Engineering Terminology" [IEEE 610.12, 1990], reaffirmed in 2002 [IEEE Standards, 2003]. The word **defect** is a generic term that refers to a fault, failure, or error. In the interests of precision, in this book we therefore minimize use of the umbrella term *defect*.

One term that is avoided as far as possible is **bug** (the history of this word is in Just in Case You Wanted to Know Box 1.7). The term *bug* nowadays is simply a euphemism for a *fault*. Although there generally is no real harm in using euphemisms, the word *bug* has overtones that are not conducive to good software production. Specifically, instead of saying, "I made a mistake," a programmer will say, "A bug crept into the code" (not *my* code but *the* code), thereby transferring responsibility for the mistake from the programmer to the bug. No one blames a programmer for coming down with a case of influenza, because the flu is caused by the flu bug. Referring to a mistake as a bug is a way of casting off responsibility. In contrast, the programmer who says, "I made a mistake," is a computer professional who takes responsibility for his or her actions.

Considerable confusion surrounds object-oriented terminology. For example, in addition to the term **attribute** for a data component of an object, the term **state variable** sometimes is used in the object-oriented literature. In Java, the term is **instance variable**. In C++ the term **field** is used, and in Visual Basic .NET, the term is **property**. With regard to the implementation of the operations of an object, the term *method* usually is used; in

C++, however, the term is **member function**. In C++, a *member* of an object refers to either an attribute ("field") or a method. In Java, the term *field* is used to denote either an attribute ("instance variable") or a method. To avoid confusion, wherever possible, the generic terms *attribute* and *method* are used in this book.

Fortunately, some terminology is widely accepted. For example, when a method within an object is invoked, this almost universally is termed **sending a message** to the object.

1.12 Ethical Issues

We conclude this chapter on a cautionary note. Software products are developed and maintained by humans. If those individuals are hard working, intelligent, sensible, up to date, and above all, *ethical*, then the chances are good that the way that the software products they develop and maintain will be satisfactory. Unfortunately, the converse is equally true.

Most societies for professionals have a code of **ethics** to which all its members must adhere. The two major societies for computer professionals, the Association for Computing Machinery (ACM) and the Computer Society of the Institute of Electrical and Electronics Engineers (IEEE-CS) jointly approved a Software Engineering Code of Ethics and Professional Practice as the standard for teaching and practicing software engineering [IEEE/ACM, 1999]. It is lengthy, so a short version, consisting of a preamble and eight principles, was also produced. Here is the short version:

Software Engineering Code of Ethics and Professional Practice[2] (Version 5.2)

as recommended by the IEEE-CS/ACM Joint Task Force on
Software Engineering Ethics and Professional Practices
Short Version
Preamble

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

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

1. *Public*—Software engineers shall act consistently with the public interest.
2. *Client and Employer*—Software engineers shall act in a manner that is in the best interests of their client and employer consistent with the public interest.

[2]© 1999 by the Institute of Electrical and Electronics Engineers, Inc., and the Association for Computing Machinery, Inc.

3. *Product*—Software engineers shall ensure that their products and related modifications meet the highest professional standards possible.
4. *Judgment*—Software engineers shall maintain integrity and independence in their professional judgment.
5. *Management*—Software engineering managers and leaders shall subscribe to and promote an ethical approach to the management of software development and maintenance.
6. *Profession*—Software engineers shall advance the integrity and reputation of the profession consistent with the public interest.
7. *Colleagues*—Software engineers shall be fair to and supportive of their colleagues.
8. *Self*—Software engineers shall participate in lifelong learning regarding the practice of their profession and shall promote an ethical approach to the practice of the profession.

The codes of ethics of other societies for computer professionals express similar sentiments. It is vital for the future of our profession that we adhere rigorously to such codes of ethics.

In Chapter 2, we examine various life-cycle models to shed further light on the differences between the classical and the object-oriented paradigm.

Chapter Review

Software engineering is defined (Section 1.1) as a discipline whose aim is the production of fault-free software that satisfies the user's needs and is delivered on time and within budget. To achieve this goal, appropriate techniques have to be used throughout software production, including when performing analysis (specification) and design (Section 1.4) and postdelivery maintenance (Section 1.3). Software engineering addresses all the steps of the software life cycle and incorporates aspects of many different areas of human knowledge, including economics (Section 1.2) and the social sciences (Section 1.5). There is no separate planning phase (Section 1.6), no testing phase (Section 1.7), and no documentation phase (Section 1.8). In Section 1.9, objects are introduced, and a comparison between the classical and object-oriented paradigms is made. Then the object-oriented paradigm is evaluated (Section 1.10). Next, in Section 1.11, the terminology used in this book is explained. Finally, ethical issues are discussed in Section 1.12.

For Further Reading

The earliest source of information on the scope of software engineering is [Boehm, 1976]. The future of software engineering is discussed in [Finkelstein, 2000]. The current state of the practice of software engineering is described in a variety of articles in the November–December 2003 issue of *IEEE Software.* An investigation of the factors leading to successful software development appears in [Procaccino, Verner, and Lorenzet, 2006].

For a view on the importance of postdelivery maintenance in software engineering and how to plan for it, see [Parnas, 1994]. Software development for COTS-based products is the subject of [Brownsword, Oberndorf, and Sledge, 2000]. Acquiring COTS components is described in [Ulkuniemi and Seppanen, 2004] and in [Keil and Tiwana, 2005]. Risk management when software is developed using COTS components is described in [Li et al., 2008]. The July–August 2005 issue of *IEEE Software* contains six articles on integrating COTS components into software products, including [Donzelli et al., 2005] and [Yang, Bhuta, Boehm, and Port, 2005]. A reassessment of risk management appears in [Bannerman, 2008].

Risks in enterprise systems are described in [Scott and Vessey, 2002] and in information systems in general in [Longstaff, Chittister, Pethia, and Haimes, 2000]. Zvegintzov [1998] explains just how little accurate data on software engineering practice actually are available.

The fact that mathematics underpins software engineering is stressed in [Devlin, 2001]. The importance of economics in software engineering is discussed in [Boehm and Huang, 2003]. The November–December 2002 issue of *IEEE Software* contains a number of articles on software engineering economics.

Two classic books on the social sciences and software engineering are [Weinberg, 1971] and [Shneiderman, 1980]. Neither book requires prior knowledge of psychology or the behavioral sciences in general.

Brooks's [1975] timeless work, *The Mythical Man-Month*, is a highly recommended introduction to the realities of software engineering. The book includes material on all the topics mentioned in this chapter.

An excellent introduction to open-source software is [Raymond, 2000]. Paulsen, Succi, and Eberlein [2004] present an empirical study comparing open- and closed-source software products. Reuse of open-source components is described in [Madanmohan and De', 2004]. A variety of articles on open-source software appears in the January/February 2004 issue of *IEEE Software* and in issue No. 2, 2005, of *IBM Systems Journal*. The issue of whether open-source software leads to increased security is discussed in [Hoepman and Jacobs, 2007]. The interplay between business and open-source software is the subject of [Watson et al., 2008], [Ven, Verelst, and Mannaert, 2008], and [Wesselius, 2008].

An excellent introduction to the object-oriented paradigm is [Budd, 2002]. Three successful projects carried out using the object-oriented paradigm are described in [Capper, Colgate, Hunter, and James, 1994], with a detailed analysis. A survey of the attitudes of 150 experienced software developers toward the object-oriented paradigm is reported in [Johnson, 2000]. With regard to ethics, an ethical code common to both business and software professionals is presented in [Payne and Landry, 2006].

Key Terms

acceptance testing *7*
adaptive maintenance *8*
analysis phase *7*
architectural design *7*
artifact *18*
attribute *25*
bug *25*
classical paradigm *18*
clickware *23*
client *23*
coding *7*
commercial-off-the-shelf (COTS) software *23*
contract software *23*
corrective maintenance *8*
defect *25*
design by contract *20*
design document *7*
design phase *7*
detailed design *7*
developer *23*
development-then-maintenance model *9*
documentation phase *17*
encapsulation *20*
enhancement *8*
error *25*
ethics *26*
failure *25*
fault *25*
field *25*
implementation phase *7*
instance variable *25*
integration *7*
internal software development *23*
life cycle *6*
life-cycle model *6*
Linus's Law *23*
maintenance *10*
message *19*
member function *26*
method *19*
methodology *24*
mistake *25*
module *7*
object-oriented paradigm *25*
open-source software *23*
operational definition (of maintenance) *10*
paradigm *24*
perfective maintenance *8*
phase *6*
planning phase *16*
postdelivery maintenance *7*
process *5*
product *24*
program *24*
property *25*

quality *17*
regression fault *20*
requirements phase *7*
responsibility-driven design *20*
retirement *8*
send a message *26*
shrink-wrapped software *23*
software *24*
software crisis *4*
software depression *5*
software engineering *2*
software project management plan *7*
software repair *8*
specification document *7*
specification phase *7*
state variable *25*
structured paradigm *18*
system *24*
systems analysis *24*
systems design *24*
technique *25*
temporal definition (of maintenance) *9*
testing phase *17*
traditional paradigm *25*
unit testing *7*
user *23*
validation *17*
verification *17*
waterfall model *7*

Problems

1.1 You are in charge of automating a multi-site architectural practice. The cost of developing the software has been estimated to be $530,000. Approximately how much additional money will be needed for postdelivery maintenance of the software?

1.2 Is there a way of reconciling the classical temporal definition of maintenance with the operational definition we now use? Explain your answer.

1.3 You are a software-engineering consultant. The chief information officer of a regional gasoline distribution corporation wants you to develop a software product that will carry out all the accounting functions of the company and provide online information to the head office staff regarding orders and inventory in the various company storage tanks. Computers are required for 21 accounting clerks, 15 order clerks, and 37 storage tank clerks. In addition, 14 managers need access to the data. The company is willing to pay $30,000 for the hardware and the software together and wants the complete software product in 4 weeks. What do you tell him? Bear in mind that your company wants his corporation's business, no matter how unreasonable his request.

1.4 You are a vice-admiral in the Velorian Navy. It has been decided to call in a software development organization to develop the control software for a new generation of ship-to-ship missiles. You are in charge of supervising the project. To protect the government of Veloria, what clauses do you include in the contract with the software developers?

1.5 You are a software engineer whose job is to supervise the development of the software in Problem 1.4. List ways your company can fail to satisfy the contract with the navy. What are the probable causes of such failures?

1.6 Nine months after delivery, a fault is detected in the software of a product that analyzes mRNA using the Stein–Röntgen reagent. The cost of fixing the fault is $18,900. The cause of the fault is an ambiguous sentence in the specification document. Approximately how much would it have cost to correct the fault during the analysis phase?

1.7 Suppose that the fault in Problem 1.6 had been detected during the implementation phase. Approximately how much would it have cost to fix then?

1.8 You are the president of an organization that builds large-scale software. You show Figure 1.6 to your employees, urging them to find faults early in the software life cycle. Someone responds that it is unreasonable to expect anyone to remove faults before they have entered the product. For example, how can anyone remove a fault while the design is being produced if the fault in question is a coding fault? What do you reply?

1.9 Describe a situation in which the client, developer, and user are the same person.

1.10 What problems can arise if the client, developer, and user are the same person? How can these problems be solved?

1.11 What potential advantages accrue if the client, developer, and user are the same person?

1.12 Look up the word *system* in a dictionary. How many different definitions are there? Write down those definitions that are applicable within the context of software engineering.

1.13 It is your first day at your first job. Your manager hands you a program listing and says, "See if you can find the bug." What do you reply?

1.14 You are in charge of developing the product in Problem 1.1. Will you use the object-oriented paradigm or the classical paradigm? Give reasons for your answer.

1.15 Instead of implementing component `c9` of a software product, the developers decide to buy a COTS component with the same specifications as component `c9`. What are the advantages and disadvantages of this approach?

1.16 Instead of implementing component `c37` of a software product, the developers decide to utilize an open-source component with the same specifications as component `c37`. What are the advantages and disadvantages of this approach?

1.17 Object `P` invokes method `m1` of object `Q`. Suppose we wish to reuse object `P` in a new software product. Can `P` be reused without reusing `Q` as well? What does this say about objects as "independent entities" (as stated in Section 1.9)?

1.18 Is it correct to state that, as a consequence of Linus's Law, all open-source software is of high quality?

1.19 (Term Project) Suppose that the product for Chocoholics Anonymous of Appendix A has been implemented exactly as described. Now the product has to be modified to include endocrinologists as providers. In what ways will the existing product have to be changed? Would it be better to discard everything and start again from scratch?

1.20 (Readings in Software Engineering) Your instructor will distribute copies of Schach et al. [2003]. What is your opinion of the relative merits of results based on managers' estimates compared to results computed from actual data?

References

[Bannerman, 2008] P. L. Bannerman, "Risk and Risk Management in Software Projects: A Reassessment," *Journal of Systems and Software* **81** (December 2008), pp. 2118–33.

[Boehm, 1976] B. W. Boehm, "Software Engineering," *IEEE Transactions on Computers* **C-25** (December 1976), pp. 1226–41.

[Boehm, 1979] B. W. Boehm, "Software Engineering, R & D Trends and Defense Needs," in: *Research Directions in Software Technology*, P. Wegner (Editor), The MIT Press, Cambridge, MA, 1979.

[Boehm, 1980] B. W. Boehm, "Developing Small-Scale Application Software Products: Some Experimental Results," *Proceedings of the Eighth IFIP World Computer Congress,* October 1980, IFIP, pp. 321–26.

[Boehm, 1981] B. W. Boehm, *Software Engineering Economics,* Prentice Hall, Englewood Cliffs, NJ, 1981.

[Boehm and Huang, 2003] B. Boehm and L. G. Huang, "Value-Based Software Engineering: A Case Study," *IEEE Computer* **36** (March 2003), pp. 33–41.

[Brooks, 1975] F. P. Brooks, Jr., *The Mythical Man-Month: Essays on Software Engineering,* Addison-Wesley, Reading, MA, 1975; Twentieth Anniversary Edition, Addison-Wesley, Reading, MA, 1995.

[Brownsword, Oberndorf, and Sledge, 2000] L. Brownsword, T. Oberndorf, and C. A. Sledge, "Developing New Process for COTS-Based Systems," *IEEE Software* **17** (July–August 2000), pp. 40–47.

[Budd, 2002] T. A. Budd, *An Introduction to Object-Oriented Programming*, 3rd ed., Addison-Wesley, Reading, MA, 2002.

[Capper, Colgate, Hunter, and James, 1994] N. P. Capper, R. J. Colgate, J. C. Hunter, and M. F. James, "The Impact of Object-Oriented Technology on Software Quality: Three Case Histories," *IBM Systems Journal* **33** (No. 1, 1994), pp. 131–57.

[Cutter Consortium, 2002] Cutter Consortium, "78% of IT Organizations Have Litigated," *The Cutter Edge*, www.cutter.com/research/2002/edge020409.html,[3] April 09, 2002.

[Daly, 1977] E. B. Daly, "Management of Software Development," *IEEE Transactions on Software Engineering* **SE-3** (May 1977), pp. 229–42.

[Devlin, 2001] K. Devlin, "The Real Reason Why Software Engineers Need Math," *Communications of the ACM* **44** (October 2001), pp. 21–22.

[Donzelli et al., 2005] P. Donzelli, M. Zelkowitz, V. Basili, D. Allard, and K. N. Meyer, "Evaluating COTS Component Dependability in Context," *IEEE Software* **22** (July–August 2005), pp. 46–53.

[Elshoff, 1976] J. L. Elshoff, "An Analysis of Some Commercial PL/I Programs," *IEEE Transactions on Software Engineering* **SE-2** (June 1976), pp. 113–20.

[Fagan, 1974] M. E. Fagan, "Design and Code Inspections and Process Control in the Development of Programs," Technical Report IBM-SSD TR 21.572, IBM Corporation, December 1974.

[Finkelstein, 2000] A. Finkelstein (Editor), *The Future of Software Engineering*, IEEE Computer Society Press, Los Alamitos, CA, 2000.

[GJSentinel.com, 2003] "Sallie Mae's Errors Double Some Bills," www.gjsentinel.com/news/content/coxnet/headlines/0522_salliemae.html, May 22, 2003.

[Grady, 1994] R. B. Grady, "Successfully Applying Software Metrics," *IEEE Computer* **27** (September 1994), pp. 18–25.

[Hatton, 1998] L. Hatton, "Does OO Sync with How We Think?" *IEEE Software* **15** (May–June 1998), pp. 46–54.

[Hoepman and Jacobs, 2007] J.-H. Hoepman and B. Jacobs, "Increased Security through Open Source," *Communications of the ACM* **50** (January 2007), pp. 79–83.

[IEEE 610.12, 1990] "A Glossary of Software Engineering Terminology," IEEE 610.12-1990, Institute of Electrical and Electronic Engineers, Inc., 1990.

[IEEE Standards, 2003] "Products and Projects Status Report," standards.ieee.org/db/status/status.txt, June 3, 2003.

[IEEE/ACM, 1999] "Software Engineering Code of Ethics and Professional Practice, Version 5.2, as Recommended by the IEEE-CS/ACM Joint Task Force on Software Engineering Ethics and Professional Practice," www.computer.org/tab/seprof/code.htm, 1999.

[IEEE/EIA 12207.0-1996, 1998] "IEEE/EIA 12207.0-1996 Industry Implementation of International Standard ISO/IEC 12207:1995," Institute of Electrical and Electronic Engineers, Electronic Industries Alliance, New York, 1998.

[ISO/IEC 12207, 1995] "ISO/IEC 12207:1995, Information Technology—Software Life-Cycle Processes," International Organization for Standardization, International Electrotechnical Commission, Geneva, 1995.

[3]This and the other URLs cited in this book were correct at the time of going to press. However, Web addresses tend to change all too frequently and without prior or subsequent notification. If this happens, the reader should use a search engine to locate the new URL. The date given in a reference to a URL is the publication date.

[Johnson, 2000] R. A. JOHNSON, "The Ups and Downs of Object-Oriented System Development," *Communications of the ACM* **43** (October 2000), pp. 69–73.

[Josephson, 1992] M. JOSEPHSON, *Edison, A Biography*, John Wiley and Sons, New York, 1992.

[Kan et al., 1994] S. H. KAN, S. D. DULL, D. N. AMUNDSON, R. J. LINDNER, AND R. J. HEDGER, "AS/400 Software Quality Management," *IBM Systems Journal* **33** (No. 1, 1994), pp. 62–88.

[Keil and Tiwana, 2005] M. KEIL AND A. TIWANA, "Beyond Cost: The Drivers of COTS Application Value," *IEEE Software* **22** (May–June 2005), pp. 64–69.

[Kelly, Sherif, and Hops, 1992] J. C. KELLY, J. S. SHERIF, AND J. HOPS, "An Analysis of Defect Densities Found during Software Inspections," *Journal of Systems and Software* **17** (January 1992), pp. 111–17.

[La Libre Online, 2007a] "Lalibre.be—Une erreur à 883 millions d'euros," www.lalibre.be/index.php?view=article&art_id=305607.

[La Libre Online, 2007b] "Lalibre.be—C'est la faute à l'informatique," www.lalibre.be/index.php?view=article&art_id=307021.

[Leveson and Turner, 1993] N. G. LEVESON AND C. S. TURNER, "An Investigation of the Therac-25 Accidents," *IEEE Computer* **26** (July 1993), pp. 18–41.

[Li et al., 2008] J. LI, O. P. N. SLYNGSTAD, M. TORCHIANO, M. MORISIO, AND C. BUNSE, "A State-of-the-Practice Survey of Risk Management in Development with Off-the-Shelf Software Components," *IEEE Transactions on Software Engineering* **34** (March–April 2008), pp. 271–86.

[Lientz, Swanson, and Tompkins, 1978] B. P. LIENTZ, E. B. SWANSON, AND G. E. TOMPKINS, "Characteristics of Application Software Maintenance," *Communications of the ACM* **21** (June 1978), pp. 466–71.

[Longstaff, Chittister, Pethia, and Haimes, 2000] T. A. LONGSTAFF, C. CHITTISTER, R. PETHIA, AND Y. Y. HAIMES, "Are We Forgetting the Risks of Information Technology?" *IEEE Computer* **33** (December 2000), pp. 43–51.

[Madanmohan and De', 2004] T. R. MADANMOHAN AND R. DE', "Open Source Reuse in Commercial Firms," *IEEE Software* **21** (November–December 2004), pp. 62–69.

[Mellor, 1994] P. MELLOR, "CAD: Computer-Aided Disaster," Technical Report, Centre for Software Reliability, City University, London, July 1994.

[Meyer, 1992] B. MEYER, "Applying 'Design by Contract'," *IEEE Computer* **25** (October 1992), pp. 40–51.

[Naur, Randell, and Buxton, 1976] P. NAUR, B. RANDELL, AND J. N. BUXTON (Editors), *Software Engineering: Concepts and Techniques: Proceedings of the NATO Conferences*, Petrocelli-Charter, New York, 1976.

[Neumann, 1980] P. G. NEUMANN, Letter from the Editor, *ACM SIGSOFT Software Engineering Notes* **5** (July 1980), p. 2.

[Parnas, 1994] D. L. PARNAS, "Software Aging," *Proceedings of the 16th International Conference on Software Engineering*, Sorrento, Italy, May 1994, IEEE, pp. 279–87.

[Paulson, Succi, and Eberlein, 2004] J. W. PAULSON, G. SUCCI, AND A. EBERLEIN, "An Empirical Study of Open-Source and Closed-Source Software Products," *IEEE Transactions on Software Engineering* **30** (April 2004), pp. 246–56.

[Payne and Landry, 2006] D. PAYNE AND B. J. L. LANDRY, "A Uniform Code of Ethics: Business and IT Professional Ethics," *Communications of the ACM* **49** (November 2006), pp. 81–84.

[Procaccino, Verner, and Lorenzet, 2006] J. D. PROCACCINO, J. M. VERNER, AND S. J. LORENZET, "Defining and Contributing to Software Development Success," *Communications of the ACM* (August 2006), pp. 79–83.