1

Introduction

LEARNING OBJECTIVES

- To understand the notion of software engineering and why it is important
- To appreciate the technical (engineering), managerial, and psychological aspects of software engineering
- To understand the similarities and differences between software engineering and other engineering disciplines
- To know the major phases in a software development project
- To appreciate ethical dimensions in software engineering
- To be aware of the time frame and extent to which new developments impact software engineering practice

Software engineering concerns methods and techniques to develop large software systems. The engineering metaphor is used to emphasize a systematic approach to develop systems that satisfy organizational requirements and constraints. This chapter gives a brief overview of the field and points at emerging trends that influence the way software is developed.

Computer science is still a young field. The first computers were built in the mid 1940s, since when the field has developed tremendously.

Applications from the early years of computerization can be characterized as follows: the programs were quite small, certainly when compared to those that are currently being constructed; they were written by one person; they were written and used by experts in the application area concerned. The problems to be solved were mostly of a technical nature, and the emphasis was on expressing known algorithms efficiently in some programming language. Input typically consisted of numerical data, read from such media as punched tape or punched cards. The output, also numeric, was printed on paper. Programs were run off-line. If the program contained errors, the programmer studied an octal or hexadecimal dump of memory. Sometimes, the execution of the program would be followed by binary reading machine registers at the console.

Independent software development companies hardly existed in those days. Software was mostly developed by hardware vendors and given away for free. These vendors sometimes set up user groups to discuss requirements, and next incorporated them into their software. This software development support was seen as a service to their customers.

Present-day applications are rather different in many respects. Present-day programs are often very large and are being developed by teams that collaborate over periods spanning several years. These teams may be scattered across the globe. The programmers are not the future users of the system they develop and they have no expert knowledge of the application area in question. The problems that are being tackled increasingly concern everyday life: automatic bank tellers, airline reservation, salary administration, electronic commerce, automotive systems, etc. Putting a man on the moon was not conceivable without computers.

In the 1960s, people started to realize that programming techniques had lagged behind the developments in software both in size and complexity. To many people, programming was still an *art* and had never become a *craft*. An additional problem was that many programmers had not been formally educated in the field. They had learned by doing. On the organizational side, attempted solutions to problems often involved adding more and more programmers to the project, the so-called 'million-monkey' approach.

As a result, software was often delivered too late, programs did not behave as the user expected, programs were rarely adaptable to changed circumstances, and many errors were detected only after the software had been delivered to the customer. This

became known as the 'software crisis'.

This type of problem really became manifest in the 1960s. Under the auspices of NATO, two conferences were devoted to the topic in 1968 and 1969 (Naur and Randell, 1968), (Buxton and Randell, 1969). Here, the term 'software engineering' was coined in a somewhat provocative sense. Shouldn't it be possible to build software in the way one builds bridges and houses, starting from a theoretical basis and using sound and proven design and construction techniques, as in other engineering fields?

Software serves some organizational purpose. The reasons for embarking on a software development project vary. Sometimes, a solution to a problem is not feasible without the aid of computers, such as weather forecasting, or automated bank telling. Sometimes, software can be used as a vehicle for new technologies, such as typesetting, the production of chips, or manned space trips. In yet other cases software may increase user service (library automation, e-commerce) or simply save money (automated stock control).

In many cases, the expected economic gain will be a major driving force. It may not, however, always be easy to prove that automation saves money (just think of office automation) because apart from direct cost savings, the economic gain may also manifest itself in such things as a more flexible production or a faster or better user service. In either case, it is a value-creating activity.

Boehm (1981) estimated the total expenditure on software in the US to be $40 billion in 1980. This is approximately 2% of the GNP. In 1985, the total expenditure had risen to $70 billion in the US and $140 billion worldwide. Boehm and Sullivan (1999) estimated the annual expenditure on software development in 1998 to be $300-400 billion in the US, and twice that amount worlwide.

So the *cost* of software is of crucial importance. This concerns not only the cost of developing the software, but also the cost of keeping the software operational once it has been delivered to the customer. In the course of time, hardware costs have decreased dramatically. Hardware costs now typically comprise less than 20% of total expenditure (figure 1.1). The remaining 80% comprise all non-hardware costs: the cost of programmers, analysts, management, user training, secretarial help, etc.

An aspect closely linked with cost is *productivity*. In the 1980s, the quest for data processing personnel increased by 12% per year, while the population of people working in data processing and the productivity of those people each grew by approximately 4% per year (Boehm, 1987a). This situation has not fundamentally changed (Jones, 1999). The net effect is a growing gap between demand and supply. The result is both a backlog with respect to the maintenance of existing software and a slowing down in the development of new applications. The combined effect may have repercussions on the competitive edge of an organization, especially so when there are severe time-to-market constraints. These developments have led to a shift from *producing* software to *using* software. We'll come back to this topic in section 1.6 and chapter ??.

The issues of cost and productivity of software development deserve our serious attention. However, this is not the complete story. Society is increasingly dependent

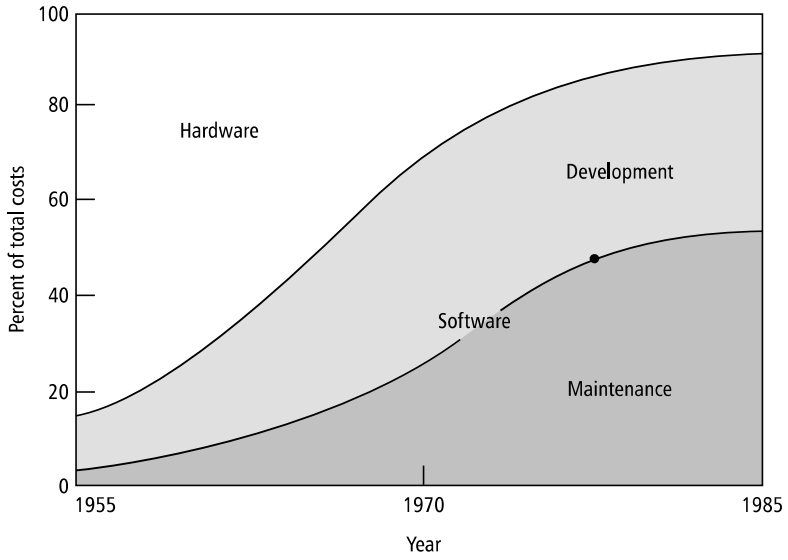

Figure 1.1 Relative distribution of hardware/software costs. (*Source: B.W. Boehm, Software Engineering*, IEEE Transactions on Computers, *1976 IEEE.*)

on software. The quality of the systems we develop increasingly determines the quality of our existence. Consider as an example the following message from a Dutch newspaper on June 6, 1980, under the heading 'Americans saw the Russians coming':

> For a short period last Tuesday the United States brought their atomic bombers and nuclear missiles to an increased state of alarm when, because of a computer error, a false alarm indicated that the Soviet Union had started a missile attack.

Efforts to repair the error were apparently in vain, for on June 9, 1980, the same newspaper reported:

> For the second time within a few days, a deranged computer reported that the Soviet Union had started a nuclear attack against the United States. Last Saturday, the DoD affirmed the false message, which resulted in the engines of the planes of the strategic air force being started.

It is not always the world that is in danger. On a smaller scale, errors in software may have very unfortunate consequences, such as transaction errors in bank traffic; reminders to finally pay that bill of $0.00; a stock control system that issues orders too late and thus lays off complete divisions of a factory.

The latter example indicates that errors in a software system may have serious financial consequences for the organization using it. One example of such a financial loss is the large US airline company that lost $50M because of an error in their seat reservation system. The system erroneously reported that cheap seats were sold out, while in fact there were plenty available. The problem was detected only after quarterly results lagged considerably behind those of both their own previous periods and those of their competitors.

Errors in automated systems may even have fatal effects. One computer science weekly magazine contained the following message in April 1983:

> The court in Düsseldorf has discharged a woman (54), who was on trial for murdering her daughter. An erroneous message from a computerized system made the insurance company inform her that she was seriously ill. She was said to suffer from an incurable form of syphilis. Moreover, she was said to have infected both her children. In panic, she strangled her 15 year old daughter and tried to kill her 13 year old son and herself. The boy escaped, and with some help he enlisted prevented the woman from dying of an overdose. The judge blamed the computer error and considered the woman not responsible for her actions.

This all marks the enormous importance of the field of software engineering. Better methods and techniques for software development may result in large financial savings, in more effective methods of software development, in systems that better fit user needs, in more reliable software systems, and thus in a more reliable environment in which those systems function. Quality and productivity are two central themes in the field of software engineering.

On the positive side, it is imperative to point to the enormous progress that has been made since the 1960s. Software is ubiquitous and scores of trustworthy systems have been built. These range from small spreadsheet applications to typesetting systems, banking systems, Web browsers and the Space Shuttle software. The techniques and methods discussed in this book have contributed their mite to the success of these and many other software development projects.

1.1 What is Software Engineering?

In various texts on this topic, one encounters a definition of the term software engineering. An early definition was given at the first NATO conference (Naur and Randell, 1968):

> Software engineering is the establishment and use of sound engineering principles in order to obtain economically software that is reliable and works efficiently on real machines.

The definition given in the *IEEE Standard Glossary of Software Engineering Terminology* (IEEE610, 1990) is as follows:

> Software engineering is the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software.

These and other definitions of the term software engineering use rather different words. However, the essential characteristics of the field are always, explicitly or implicitly, present:

- *Software engineering concerns the development of large programs.*
  (DeRemer and Kron, 1976) make a distinction between **programming-in-the-large** and **programming-in-the-small**. The borderline between large and small obviously is not sharp: a program of 100 lines is small, a program of 50 000 lines of code certainly is not. Programming-in-the-small generally refers to programs written by one person in a relatively short period of time. Programming-in-the-large, then, refers to multi-person jobs that span, say, more than half a year. For example:

  - The NASA Space Shuttle software contains 40M lines of object code (this is 30 times as much as the software for the Saturn V project from the 1960s) (Boehm, 1981);
  - The IBM OS360 operating system took 5000 man years of development effort (Brooks, 1995).

  Traditional programming techniques and tools are primarily aimed at supporting programming-in-the-small. This not only holds for programming languages, but also for the tools (like flowcharts) and methods (like structured programming). These cannot be directly transferred to the development of large programs.

  In fact, the term program -- in the sense of a self-contained piece of software that can be invoked by a user or some other system component -- is not adequate here. Present-day software development projects result in systems containing a large number of (interrelated) programs -- or components.

- *The central theme is mastering complexity.*
  In general, the problems are such that they cannot be surveyed in their entirety. One is forced to split the problem into parts such that each individual part can be grasped, while the communication between the parts remains simple. The total complexity does not decrease in this way, but it does become manageable. In a stereo system there are components such as an amplifier, a receiver, and a tuner, and communication via a thin wire. In software, we strive for a similar separation of concerns. In a program for library automation, components such as user interaction, search processes and data storage could for instance be distinguished, with clearly given facilities for data exchange between those components. Note that the complexity of many a piece of software is not

so much caused by the intrinsic complexity of the problem (as in the case of compiler optimization algorithms or numerical algorithms to solve partial differential equations), but rather by the vast number of details that must be dealt with.

- *Software evolves.*
  Most software models a part of reality, such as processing requests in a library or tracking money transfers in a bank. This reality evolves. If software is not to become obsolete fairly quickly, it has to evolve with the reality that is being modeled. This means that costs are incurred after delivery of the software system and that we have to bear this evolution in mind during development.

- *The efficiency with which software is developed is of crucial importance.*
  Total cost and development time of software projects is high. This also holds for the maintenance of software. The quest for new applications surpasses the workforce resource. The gap between supply and demand is growing. Time-to-market demands ask for quick delivery. Important themes within the field of software engineering concern better and more efficient methods and tools for the development and maintenance of software, especially methods and tools enabling the use and reuse of components.

- *Regular cooperation between people is an integral part of programming-in-the-large.*
  Since the problems are large, many people have to work concurrently at solving those problems. Increasingly often, teams at different geographic locations work together in software development. There must be clear arrangements for the distribution of work, methods of communication, responsibilities, and so on. Arrangements alone are not sufficient, though; one also has to stick to those arrangements. In order to enforce them, standards or procedures may be employed. Those procedures and standards can often be supported by tools. Discipline is one of the keys to the successful completion of a software development project.

- *The software has to support its users effectively.*
  Software is developed in order to support users at work. The functionality offered should fit users' tasks. Users that are not satisfied with the system will try to circumvent it or, at best, voice new requirements immediately. It is not sufficient to build the system in the right way, we also have to build the right system. Effective user support means that we must carefully study users at work, in order to determine the proper functional requirements, and we must address usability and other quality aspects as well, such as reliability, responsiveness, and user-friendliness. It also means that software development entails more than delivering software. User manuals and training material may have to be written, and attention must be given to developing the environment in which the new system is going to be installed. For example, a new automated library system will affect working procedures within the library.

- *Software engineering is a field in which members of one culture create artifacts on behalf of members of another culture.*
  This aspect is closely linked to the previous two items. Software engineers are expert in one or more areas such as programming in Java, software architecture, testing, or the Unified Modeling Language. They are generally not experts in library management, avionics, or banking. Yet they have to develop systems for such domains. The thin spread of application domain knowledge is a common source of problems in software development projects.

  Not only do software engineers lack factual knowledge of the domain for which they develop software, they lack knowledge of its culture as well. For example, a software developer may discover the 'official' set of work practices of a certain user community from interviews, written policies, and the like; these work practices are then built into the software. A crucial question with respect to system acceptance and success, however, is whether that community actually follows those work practices. For an outside observer, this question is much more difficult to answer.

- *Software engineering is a balancing act.*
  In most realistic cases, it is illusive to assume that the collection of requirements voiced at the start of the project is the only factor that counts. In fact, the term requirement is a misnomer. It suggests something immutable, while in fact most requirements are negotiable. There are numerous business, technical and political constraints that may influence a software development project. For example, one may decide to use database technology X rather than Y, simply because of available expertise with that technology. In extreme cases, characteristics of available components may determine functionality offered, rather than the other way around.

The above list shows that software engineering has many facets. Software engineering certainly is *not* the same as programming, although programming is an important ingredient of software engineering. Mathematical aspects play a role since we are concerned with the correctness of software. Sound engineering practices are needed to get useful products. Psychological and sociological aspects play a role in the communication between human and machine, organization and machine, and between humans. Finally, the development process needs to be controlled, which is a management issue.

The term 'software engineering' hints at possible resemblances between the construction of programs and the construction of houses or bridges. These kinds of resemblances do exist. In both cases we work from a set of desired functions, using scientific and engineering techniques in a creative way. Techniques that have been applied successfully in the construction of physical artifacts are also helpful when applied to the construction of software systems: development of the product in a number of phases, a careful planning of these phases, continuous audit of the whole process, construction from a clear and complete design, etc.

Even in a mature engineering discipline, say bridge design, accidents do happen. Bridges collapse once in a while. Most problems in bridge design occur when designers extrapolate beyond their models and expertise. A famous example is the Tacoma Narrows Bridge failure in 1940. The designers of that bridge extrapolated beyond their experience to create more flexible stiffening girders for suspension bridges. They did not think about aerodynamics and the response of the bridge to wind. As a result, that bridge collapsed shortly after it was finished. This type of extrapolation seems to be the rule rather than the exception in software development. We regularly embark on software development projects that go far beyond our expertise.

There are additional reasons for considering the construction of software as something quite different from the construction of physical products. The cost of constructing software is incurred during development and not during production. Copying software is almost free. Software is logical in nature rather than physical. Physical products wear out in time and therefore have to be maintained. Software does not wear out. The need to maintain software is caused by errors detected late or by changing requirements of the user. Software reliability is determined by the manifestation of errors already present, not by physical factors such as wear and tear. We may even argue that software wears out *because* it is being maintained.

Viewing software engineering as a branch of engineering is problematic for another reason as well. The engineering metaphor hints at disciplined work, proper planning, good management, and the like. It suggests we deal with clearly defined needs, that can be fulfilled if we follow all the right steps. Many software development projects though involve the translation of some real world phenomenon into digital form. The knowledge embedded in this real life phenomenon is tacit, undefined, uncodified, and may have developed over a long period of time. The assumption that we are dealing with a well-defined problem simply does not hold. Rather, the design process is open ended, and the solution emerges as we go along. This dichotomy is reflected in views of the field put in the forefront over time (Eischen, 2002). In the early days, the field was seen as a craft. As a countermovement, the term software engineering was coined, and many factory concepts got introduced. In the late 1990's, the pendulum swung back again and the craft aspect got emphasized anew, in the agile movement (see chapter 3). Both engineering-like and craft-like aspects have their place, and we will give a balanced treatment of both.

Two characteristics that make software development projects extra difficult to manage are visibility and continuity. It is much more difficult to see progress in software construction than it is to notice progress in building a bridge. One often hears the phrase that a program 'is almost finished'. One equally often underestimates the time needed to finish up the last bits and pieces.

This '90% complete' syndrome is very pervasive in software development. Not knowing how to measure real progress, we often use a surrogate measure, the rate of expenditure of resources. For example, a project that has a budget of 100 person-days is perceived as being 50% complete after 50 person-days are expended. Strictly speaking, we then confuse speed with progress. Because of the imprecise measurement

of progress and the customary underestimation of total effort, problems accumulate as time elapses.

Physical systems are often continuous in the sense that small changes in the specification lead to small changes in the product. This is not true with software. Small changes in the specification of software may lead to considerable changes in the software itself. In a similar way, small errors in software may have considerable effects. The Mariner space rocket to Venus for example got lost because of a typing error in a FORTRAN program. In 1998, the Mars Climate Orbiter got lost, because one development team used English units such as inches and feet, while another team used metric units.

We may likewise draw a comparison between software engineering and computer science. Computer science emerged as a separate discipline in the 1960s. It split from mathematics and has been heavily influenced by mathematics. Topics studied in computer science, such as algorithm complexity, formal languages, and the semantics of programming languages, have a strong mathematical flavor. PhD theses in computer science invariably contain theorems with accompanying proofs.

As the field of software engineering emerged from computer science, it had a similar inclination to focus on clean aspects of software development that can be formalized, in both teaching and research. We used to assume that requirements can be fully stated before the project started, concentrated on systems built from scratch, and ignored the reality of trading off quality aspects against the available budget. Not to mention the trenches of software maintenance.

Software engineering and computer science do have a considerable overlap. The practice of software engineering however also has to deal with such matters as the management of huge development projects, human factors (regarding both the development team and the prospective users of the system) and cost estimation and control. Software engineers must *engineer* software.

Software engineering has many things in common both with other fields of engineering and with computer science. It also has a face of its own in many ways.

1.2 Phases in the Development of Software

When building a house, the builder does not start with piling up bricks. Rather, the requirements and possibilities of the client are analyzed first, taking into account such factors as family structure, hobbies, finances and the like. The architect takes these factors into consideration when designing a house. Only after the design has been agreed upon is the actual construction started.

It is expedient to act in the same way when constructing software. First, the problem to be solved is analyzed and the requirements are described in a very precise way. Then a design is made based on these requirements. Finally, the construction process, i.e. the actual programming of the solution, is started. There are a distinguishable number of phases in the development of software. The phases as discussed in this book are depicted in figure 1.2.

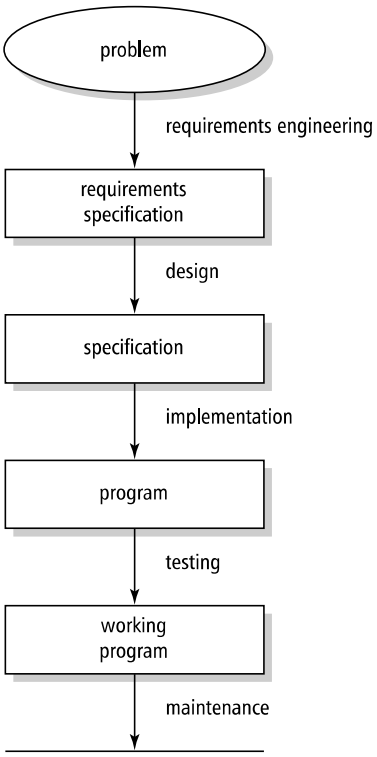

Figure 1.2 A simple view of software development

The **process model** depicted in figure 1.2 is rather simple. In reality, things will usually be more complex. For instance, the design phase is often split into a global, architectural design phase and a detailed design phase, and often various test phases are distinguished. The basic elements, however, remain as given in figure 1.2. These phases have to be passed through in each project. Depending on the kind of project and the working environment, a more detailed scheme may be needed.

In figure 1.2, the phases have been depicted sequentially. For a given project these activities are not necessarily separated as strictly as indicated here. They may and usually will overlap. It is, for instance, quite possible to start implementation of one part of the system while some of the other parts have not been fully designed yet. As we will see in section 1.3, there is no strict linear progression from requirements engineering to design, from design to implementation, etc. Backtracking to earlier phases occurs, because of errors discovered or changing requirements. One had better

think of these phases as a series of workflows. Early on, most resources are spent on the requirements engineering workflow. Later on, effort moves to the implementation and testing workflows.

Below, a short description is given of each of the basic elements from figure 1.2. Various alternative process models will be discussed in chapter 3. These alternative models result from justifiable criticism of the simple-minded model depicted in figure 1.2. The sole aim of our simple model is to provide an adequate structuring of topics to be addressed. The maintenance phase is further discussed in section 1.3. All elements of our process model will be treated much more elaborately in later chapters.

Requirements engineering. The goal of the requirements engineering phase is to get a complete description of the problem to be solved and the requirements posed by and on the environment in which the system is going to function. Requirements posed by the environment may include hardware and supporting software or the number of prospective users of the system to be developed. Alternatively, analysis of the requirements may lead to certain constraints imposed on hardware yet to be acquired or to the organization in which the system is to function. A description of the problem to be solved includes such things as:

- the functions of the software to be developed;
- possible future extensions to the system;
- the amount, and kind, of documentation required;
- response time and other performance requirements of the system.

Part of requirements engineering is a **feasibility study**. The purpose of the feasibility study is to assess whether there is a solution to the problem which is both economically and technically feasible.

The more careful we are during the requirements engineering phase, the larger is the chance that the ultimate system will meet expectations. To this end, the various people (among others, the customer, prospective users, designers, and programmers) involved have to collaborate intensively. These people often have widely different backgrounds, which does not ease communication.

The document in which the result of this activity is laid down is called the **requirements specification**.

Design. During the design phase, a model of the whole system is developed which, when encoded in some programming language, solves the problem for the user. To this end, the problem is decomposed into manageable pieces called **components**; the functions of these components and the **interfaces** between them are specified in a very precise way. The design phase is crucial. Requirements engineering and design are sometimes seen as an annoying introduction to programming, which is often seen

as the real work. This attitude has a very negative influence on the quality of the resulting software.

Early design decisions have a major impact on the quality of the final system. These early design decisions may be captured in a global description of the system, i.e. its **architecture**. The architecture may next be evaluated, serve as a template for the development of a family of similar systems, or be used as a skeleton for the development of reusable components. As such, the architectural description of a system is an important milestone document in present-day software development projects.

During the design phase we try to separate the *what* from the *how*. We concentrate on the problem and should not let ourselves be distracted by implementation concerns.

The result of the design phase, the (**technical**) **specification**, serves as a starting point for the implementation phase. If the specification is formal in nature, it can also be used to derive correctness proofs.

Implementation. During the implementation phase, we concentrate on the individual components. Our starting point is the component's specification. It is often necessary to introduce an extra 'design' phase, the step from component specification to executable code often being too large. In such cases, we may take advantage of some high-level, programming-language-like notation, such as a **pseudocode**. (A pseudocode is a kind of programming language. Its syntax and semantics are in general less strict, so that algorithms can be formulated at a higher, more abstract, level.)

It is important to note that the first goal of a programmer should be the development of a well-documented, reliable, easy to read, flexible, correct, program. The goal is *not* to produce a very efficient program full of tricks. We will come back to the many dimensions of software quality in chapter 6.

During the design phase, a global structure is imposed through the introduction of components and their interfaces. In the more classic programming languages, much of this structure tends to get lost in the transition from design to code. More recent programming languages offer possibilities to retain this structure in the final code through the concept of modules or classes.

The result of the implementation phase is an executable program.

Testing. Actually, it is wrong to say that testing is a phase following implementation. This suggests that you need not bother about testing until implementation is finished. This is not true. It is even fair to say that this is one of the biggest mistakes you can make.

Attention has to be paid to testing even during the requirements engineering phase. During the subsequent phases, testing is continued and refined. The earlier that errors are detected, the cheaper it is to correct them.

Testing at phase boundaries comes in two flavors. We have to test that the transition between subsequent phases is correct (this is known as **verification**). We also have to check that we are still on the right track as regards fulfilling user

requirements (**validation**). The result of adding verification and validation activities to the linear model of figure 1.2 yields the so-called **waterfall model** of software development (see also chapter 3).

Maintenance. After delivery of the software, there are often errors that have still gone undetected. Obviously, these errors must be repaired. In addition, the actual use of the system can lead to requests for changes and enhancements. All these types of changes are denoted by the rather unfortunate term maintenance. Maintenance thus concerns all activities needed to keep the system operational after it has been delivered to the user.

An activity spanning all phases is **project management**. Like other projects, software development projects must be managed properly in order to ensure that the product is delivered on time and within budget. The visibility and continuity characteristics of software development, as well as the fact that many software development projects are undertaken with insufficient prior experience, seriously impede project control. The many examples of software development projects that fail to meet their schedule provide ample evidence of the fact that we have by no means satisfactorily dealt with this issue yet. Chapters 2--8 deal with major aspects of software project management, such as project planning, team organization, quality issues, cost and schedule estimation.

An important activity not identified separately is **documentation**. A number of key ingredients of the documentation of a software project will be elaborated upon in the chapters to follow. Key components of system documentation include the project plan, quality plan, requirements specification, architecture description, design documentation and test plan. For larger projects, a considerable amount of effort will have to be spent on properly documenting the project. The documentation effort must start early on in the project. In practice, documentation is often seen as a balancing item. Since many projects are pressed for time, the documentation tends to get the worst of it. Software maintainers and developers know this, and adapt their way of working accordingly. As a rule of thumb, Lethbridge et al. (2003) states that, the closer one gets to the code, the more accurate the documentation must be for software engineers to use it. Outdated requirements documents and other high-level documentation may still give valuable clues. They are useful to people who have to learn about a new system or have to develop test cases, for instance. Outdated low-level documentation is worthless, and makes that programmers consult the code rather than its documentation. Since the system will undergo changes after delivery, because of errors that went undetected or changing user requirements, proper and up-to-date documentation is of crucial importance during maintenance.

A particularly noteworthy element of documentation is the user documentation. Software development should be task-oriented in the sense that the software to be delivered should support users in their task environment. Likewise, the user documentation should be task- oriented. User manuals should not just describe the features of a system, they should help people to get things done (Rettig, 1991). We

cannot simply rely on the structure of the interface to organize the user documentation (just as a programming language reference manual is not an appropriate source for learning how to program).

Figure 1.3 depicts the relative effort spent on the various activities up to delivery of the system. From this data a very clear trend emerges, the so-called 40--20--40 rule: only 20% of the effort is spent on actually programming (coding) the system, while the preceding phases (requirements engineering and design) and testing each consume about 40% of the total effort.

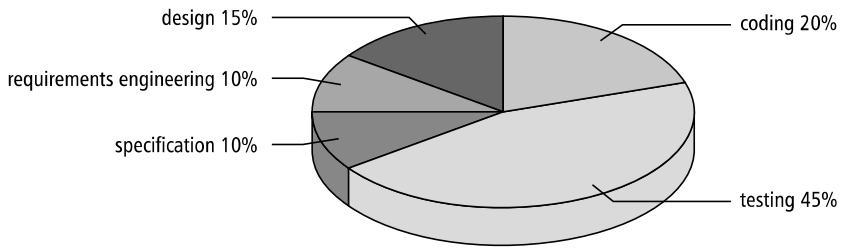

Figure 1.3 Relative effort for the various activities

Depending on specific boundary conditions, properties of the system to be constructed, and the like, variations to this rule can be found. For iterative development projects, the distinction between requirements engineering, design, implementation and (unit) testing gets blurred, for instance. For the majority of projects, however, this rule of thumb is quite workable.

This does not imply that the 40--20--40 rule is the one to be strived for. Errors made during requirements engineering are the ones that are most costly to repair (see also the chapter on testing). It is far better to put more energy into the requirements engineering phase, than to try to remove errors during the time-consuming testing phase or, worse still, during maintenance. According to (Boehm, 1987b), successful projects follow a 60--15--25 distribution: 60% requirements engineering and design, 15% implementation and 25% testing. The message is clear: the longer you postpone coding, the earlier you are finished.

Figure 1.3 does not show the extent of the maintenance effort. When we consider the total cost of a software system over its lifetime, it turns out that, on average, maintenance alone consumes 50--75% of these costs; see also figure 1.1. Thus, maintenance alone consumes more than the various development phases taken together.

1.3 Maintenance or Evolution

The only thing we maintain is user satisfaction
(Lehman, 1980)

Once software has been delivered, it usually still contains errors which, upon discovery, must be repaired. Note that this type of maintenance is not caused by wearing. Rather, it concerns repair of hidden defects. This type of repair is comparable to that encountered after a newly-built house is first occupied.

The story becomes quite different if we start talking about changes or enhancements to the system. Repainting our office or repairing a leak in the roof of our house is called maintenance. Adding a wing to our office is seldom called maintenance.

This is more than a trifling game with words. Over the total lifetime of a software system, more money is spent on maintaining that system than on initial development. If all these expenses merely concerned the repair of errors made during one of the development phases, our business would be doing very badly indeed. Fortunately, this is not the case.

We distinguish four kinds of maintenance activities:

- **corrective** maintenance -- the repair of actual errors;
- **adaptive** maintenance -- adapting the software to changes in the environment, such as new hardware or the next release of an operating or database system;
- **perfective** maintenance -- adapting the software to new or changed user requirements, such as extra functions to be provided by the system. Perfective maintenance also includes work to increase the system's performance or to enhance its user interface;
- **preventive** maintenance -- increasing the system's future maintainability. Updating documentation, adding comments, or improving the modular structure of a system are examples of preventive maintenance activities.

Only the first category may rightfully be termed maintenance. This category, however, accounts only for about a quarter of the total maintenance effort. Approximately another quarter of the maintenance effort concerns adapting software to environmental changes, while half of the maintenance cost is spent on changes to accommodate changing user requirements, i.e. enhancements to the system (see figure 1.4).

Changes in both the system's environment and user requirements are inevitable. Software models part of reality, and reality changes, whether we like it or not. So the software has to change too. It *has* to evolve. A large percentage of what we are used to calling maintenance is actually evolution. Maintenance because of new user requirements occurs in both high and low quality systems. A successful system calls for new, unforeseen functionality, because of its use by many satisfied users. A less successful system has to be adapted in order to satisfy its customers.

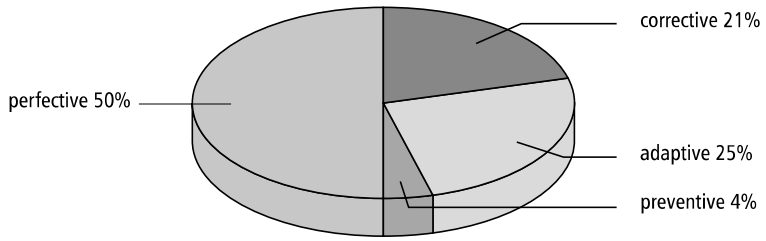

Figure 1.4 Distribution of maintenance activities

The result is that the software development process becomes cyclic, hence the phrase **software life cycle**. Backtracking to previous phases, alluded to above, does not only occur during maintenance. During other phases, also, we will from time to time iterate earlier phases. During design, it may be discovered that the requirements specification is not complete or contains conflicting requirements. During testing, errors introduced in the implementation or design phase may crop up. In these and similar cases an iteration of earlier phases is needed. We will come back to this cyclic nature of the software development process in chapter 3, when we discuss alternative models of the software development process.

1.4 From the Trenches

> *And such is the way of all superstition, whether in astrology, dreams, omens, divine judgments or the like; wherein men, having a delight in such vanities, mark the events when they are fulfilled, but when they fail, though this happens much oftener, neglect and pass them by. But with far more subtlety does this mischief insinuate itself into philosophy and the sciences; in which the first conclusion colours and brings into conformity with itself all that come after, though far sounder and better. Besides, independently of that delight and vanity which I have described, it is the peculiar and perpetual error of the human intellect to be more moved and excited by affirmatives than by negatives; whereas it ought properly to hold itself indifferently disposed towards both alike. Indeed in the establishment of any true axiom, the negative instance is the more forcible of the two.*
> Sir Francis Bacon, The New Organon, Aphorisms XLVI (1611)

> *Historical case studies contain a wealth of wisdom about the nature of design and the engineering method.*
> (Petroski, 1994)

In his wonderful book *Design Paradigms, Case Histories of Error and Judgment in Engineering*, Henri Petroski tells us about some of the greatest engineering successes and, especially,

failures of all time. Some such failure stories about our profession have appeared as well. Four of them are discussed in this section.

These stories are interesting because they teach us that software engineering has many facets. Failures in software development projects often are not one-dimensional. They are not *only* caused by a technical slip in some routine. They are not *only* caused by bad management. They are not *only* the result of human communication problems. It is often a combination of many smaller slips, which accumulate over time, and eventually result in a major failure. To paraphrase a famous saying of Fred Brooks about projects getting late:

> 'How does a project really get into trouble?'
> 'One slip at a time.'

Each of the stories discussed below shows such a cumulative effect. Successes in software development will not come about if we just employ the brightest programmers. Or apply the newest design philosophy. Or have the most extensive user consultation. Or even hire the best manager. You have to do all of that. And even more.

1.4.1 Ariane 5, Flight 501

The maiden flight of the Ariane 5 launcher took place on June 4, 1996. After about 40 seconds, at an altitude of less than 4 kilometers, the launcher broke up and exploded. This $500M loss was ultimately caused by an overflow in the conversion from a 64-bit floating point number to a 16-bit signed integer. From a software engineering point of view, the Ariane 5 story is interesting because the failure can be attributed to different causes, at different levels of understanding: inadequate testing, wrong type of reuse, or a wrong design philosophy.

The altitude of the launcher and its movements in space are measured by an Inertial Reference System (SRI -- Système de Référence Inertielle). There are two SRIs operating in parallel. Their hardware and software is identical. Most of the hardware and software for the SRI was retained from the Ariane 4. The fatal conversion took place in a piece of software in the SRI which is only meaningful before lift-off. Though this part of the software serves no purpose after the rocket has been launched, it keeps running for an additional number of seconds. This requirement was stated more than 10 years earlier for a somewhat peculiar reason. It allows for a quick restart of the countdown, in the case that it is interrupted close to lift-off. This requirement does not apply to the Ariane 5, but the software was left unchanged -- after all, it worked. Since the Ariane 5 is much faster than the Ariane 4, the rocket reaches a much higher horizontal velocity within this short period after lift-off, resulting in the above-mentioned overflow. Because of this overflow, the first SRI ceased to function. The second SRI was then activated, but since the hardware and software of both SRIs are identical, the second SRI failed as well. As a consequence, wrong data were transmitted from the SRI to the on-board computer.

On the basis of these wrong data, full nozzle deflections were commanded. These caused a very high aerodynamic load which led to the separation of the boosters from the main rocket. And this in turn triggered the self-destruction of the launcher.

There are several levels at which the Ariane 5 failure can be understood and explained:

- It was a software failure, which could have been revealed with more extensive testing. This is true: the committee investigating the event managed to expose the failure using extensive simulations.
- The failure was caused by reusing a flawed component. This is true as well but, because of physical characteristics of the Ariane 4, this flaw had never become apparent. There had been many successful Ariane 4 flights, using essentially the same SRI subsystem. Apparently, reuse is not compositional: the successful use of a component in one environment is no guarantee for successful reuse of that component in another environment.
- The failure was caused by a flaw in the design. The Ariane software follows a typical hardware design philosophy: if a component breaks down, the cause is assumed to be random and it is handled by shutting down that part and invoking a backup component. In the case of a software failure, which is not random, an identical backup is of little use. For the software part, a different line might have been followed. For instance, the component could be asked to give its best estimate of the required information.

1.4.2 Therac-25

The Therac-25 is a computer-controlled radiation machine. It has three modes:

- field-light mode. This position merely facilitates the correct positioning of the patient.
- electron mode. In electron therapy, the computer controls the (variable) beam energy and current, and magnets spread the beam to a safe concentration.
- photon (X-ray) mode. In photon mode, the beam energy is fixed. A 'beam flattener' is put between the accelerator and the patient to produce a uniform treatment field. A very high current (some 100 times higher than in electron mode) is required on one side of the beam flattener to produce a reasonable treatment dose at the other side.

The machine has a turntable which rotates the necessary equipment into position. The basic hazardous situation is obvious from the above: a photon beam is issued by the accelerator, while the beam flattener is not in position. The patient is then treated with a dose which is far too high. This happened several times. As a consequence, several patients have died and others have been seriously injured.

One of the malfunctions of the Therac-25 has become known as 'Malfunction 54'. A patient was set up for treatment. The operator keyed in the necessary data on the console in an adjacent room. While doing so, he made a mistake: he typed 'x' (for X-ray mode) instead of 'e' (for electron mode). He corrected his mistake by moving the cursor up to the appropriate field, typing in the correct code and pressing the return key a number of times until the cursor was on the command line again. He then pressed 'B' (beam on). The machine stopped and issued the message 'Malfunction 54'. This particular error message indicates a wrong dose, either too high or too low. The console indicated a substantial underdose. The operator knew that the machine often had quirks, and that these could usually be solved by simply pressing 'P' (proceed). So he did. The same error message appeared again. Normally, the operator would have audio and video contact with the patient in the treatment room. Not this time, though: the audio was broken and the video had been turned off. It was later estimated that the patient had received 16 000--25 000 rad on a very small surface, instead of the intended dose of 180 rad. The patient became seriously ill and died five months later.

The cause of this hazardous event was traced back to the software operating the radiation machine. After the operator has finished data entry, the physical set up of the machine may begin. The bending of the magnets takes about eight seconds. After the magnets are put into position, it again checks if anything has changed. If the operator manages to make changes **and** return the cursor to the command line position within the eight seconds it takes to set the magnets, part of these changes will result in changes in internal system parameters, but the system nevertheless 'thinks' that nothing has happened and simply continues. With the consequences as described above.

Accidents like this get reported to the Federal Drugs Administration (FDA). The FDA requested the manufacturer to take appropriate measures. The 'fix' suggested was as follows:

> Effective immediately, and until further notice, the key used for moving the cursor back through the prescription sequence (i.e. cursor 'UP' inscribed with an upward pointing arrow) must not be used for editing or any other purpose.
>
> To avoid accidental use of this key, the key cap must be removed and the switch contacts fixed in the open position with electrical tape or other insulating material. . . .
>
> Disabling this key means that if any prescription data entered is incorrect then an 'R' reset command must be used and the whole prescription reentered.

The FDA did not buy this remedy. In particular, they judged the tone of the notification not commensurate with the urgency for doing so. The discussion between the FDA and the manufacturer continued for quite some time before an adequate response was given to this and other failures of the Therac-25.

The Therac-25 machine and its software evolved from earlier models that were less sophisticated. In earlier versions of the software, for example, it was not possible to move up and down the screen to change individual fields. Operators noticed that different treatments often required almost the same data, which had to be keyed in all over again. To enhance usability, the feature to move the cursor around and change individual fields was added. Apparently, user friendliness may conflict with safety.

In earlier models also, the correct position of the turntable and other equipment was ensured by simple electromechanical interlocks. These interlocks are a common mechanism to ensure safety. For instance, they are used in lifts to make sure that the doors cannot be opened if the lift is in between floors. In the Therac-25, these mechanical safety devices were replaced by software. The software was thus made into a single point of failure. This overconfidence in software contributed to the Therac-25 accidents, together with inadequate software engineering practices and an inadequate reaction of management to incidents.

1.4.3 The London Ambulance Service

The London Ambulance Service (LAS) handles the ambulance traffic in Greater London. It covers an area of over 600 square miles and carries over 5000 patients per day in 750 vehicles. The LAS receives over 2000 phone calls per day, including more than 1300 emergency calls. The system we discuss here is a computer-aided dispatch (CAD) system. Such a CAD system has the following functionality:

- it handles call taking, accepts and verifies incident details including the location of the incident;
- it determines which ambulance to send;
- it handles the mobilization of the ambulance and communicates the details of the incident to the ambulance;
- it takes care of ambulance resource management, in particular the positioning of vehicles to minimize response times.

A fully-fledged CAD system is quite complex. In panic, someone might call and say that an accident has happened in front of Foyle's, assuming that everyone knows where this bookshop is located. An extensive gazetteer component including a public telephone identification helps in solving this type of problem. The CAD system also contains a radio system, mobile terminals in the ambulances, and an automatic vehicle location system.

The CAD project of the London Ambulance Service was started in the autumn of 1990. The delivery was scheduled for January 1992. At that time, however, the software was still far from complete. Over the first nine months of 1992, the system was installed piecemeal across a number of different LAS divisions, but it was never stable. On 26 and 27 October 1992, there were serious problems with the system and

it was decided to revert to a semi-manual mode of operation. On 4 November 1992, the system crashed. The Regional Health Authority established an Inquiry Team to investigate the failures and the history that led to them. They came up with an 80-page report, which reads like a suspense novel. Below, we highlight some of the issues raised in this report.

The envisaged CAD system would be a major undertaking. No other emergency service had attempted to go as far. The plan was to move from a wholly manual process -- in which forms were filled in and transported from one employee to the next via a conveyor belt -- to complete automation, in one shot. The scheme was very ambitious. The participants seem not to have fully realized the risks they were taking.

Way before the project actually started, a management consultant firm had already been asked for advice. They suggested that a packaged solution would cost $1.5M and take 19 months. Their report also stated that if a package solution could not be found, the estimates should be significantly increased. Eventually, a non-package solution was chosen, but only the numbers from this report were remembered, or so it seems.

The advertisement resulted in replies from 35 companies. The specification and timetable were next discussed with these companies. The proposed timetable was 11 months (this is not a typo). Though many suppliers raised concerns about the timetable, they were told that it was non-negotiable. Eventually, 17 suppliers provided full proposals. The lowest tender, at approximately $1M, was selected. This tender was about $700 000 cheaper than the next lowest bid. No one seems to have questioned this huge difference. The proposal selected superficially suggests that the company had experience in designing systems for emergency services. This was not a lie: they had developed administrative systems for such services. The LAS system also was far larger than anything they had previously handled.

The proposed system would impact quite significantly on the way ambulance crews carried out their jobs. It would therefore be paramount to have their full cooperation. If the crews did not press the right buttons at the right time and in the right order, chaos could result. Yet, there was very little user involvement during the requirements engineering process.

The intended CAD system would operate in an absolutely objective and impartial way and would always mobilize the optimum resource to any incident. This would overcome many of the then present working practices which management considered outmoded and not in the interest of LAS. For instance, the new system would allocate the nearest available resource regardless of the originating station. The following scenario may result:

- John's crew has to go to an accident a few miles east of their home base.

- Once there, they are directed to a hospital a few miles further east to deliver the patient.

- Another call comes in and John happens to be nearest. He is ordered to travel yet a few miles further east.

- And so on.

In this way, crews may have to operate further and further away from their home base, and in unfamiliar territory. They lose time, because they take wrong turns, or may even have to stop to ask for directions. They also have further to travel to reach their home station at the end of a shift. Crews didn't like this aspect of the new system.

The new system also took away the flexibility local emergency stations had in deciding which resource to allocate. In the new scheme, resource management was fully centralized and handled by the system. So, suppose John runs down to where the ambulances are parked and the computer has ordered him to take car number 5. John is in a hurry and maybe he cannot quickly spot car number 5, or maybe it is parked behind some other cars. So John thinks about this patient waiting for him and decides to take car number 4 instead. This means trouble.

The people responsible for those requirements were misguided or naive in believing that computer systems in themselves can bring about such changes in human practices. Computers are there to help people do their job, not vice versa. Operational straitjackets are doomed to fail.

The eventual crash on 4 November 1992 was caused by a minor programming error. Some three weeks earlier, a programmer had been working on part of the system and forgot to remove a small piece of program text. The code in itself did no harm. However, it did allocate a small amount of memory every time a vehicle mobilization was generated by the system. This memory was not deallocated. After three weeks, all memory was used up and the system crashed.

The LAS project as a whole did not fail because of this programmer mistake. That was just the last straw. The project schedule was far too tight. Management of both the London Ambulance Service and the contractor had little or no experience with software development projects of this size and complexity. They were far too optimistic in their assessment of risks. They assumed that all the people who would interact with the system, would do so in exactly the right way, all of the time. They assumed the hardware parts of the system would work exactly as specified. Management decided on the functionality of the system, with hardly any consultation with the people that would be its primary users. Any project with such characteristics is doomed to fail. From the very first day.

1.4.4 Who Counts the Votes?

It's not who votes that counts, it's who counts the votes
Josef Stalin

Traditional, non-automated election systems leave a paper trail that can be used for auditing purposes: have all votes been counted, have they been counted correctly.

Such an audit is done by an independent party. These safeguards serve to build trust in the outcome.

But what if these elections are supported by computers? As a voter, you then simply press a button. But what next? The recording and counting is hidden. How do you know your vote is not tinkered with? How can fraud be avoided? For the individual, one then needs a voter ballot, for instance a piece of paper similar to an ATM receipt, that serves to verify the voter's choice. The ballots of all voters may next be used in an independent audit of the election outcome. Most automated election systems of today do not provide these safeguards.

What if we go one step further, and provide our voters with a web application to place their votes? Below is a story about a real system of this kind. The application was developed in Java. Due to governmental regulations, the voting model implemented mimicked the traditional one. The application maintains a voting register containing identifications of all voters, and a ballot box in which the votes are stored. One of the regulations that the system had to comply with is anonymity: a vote in the ballot box should not be traceable to a name in the voters' register. Another regulation concenrs security: both registers have to be stored separately.

The technical design envisaged two separate databases, one for the voters and one for the ballots. Placing a vote and marking a voter as 'has voted' should be performed in a single transaction: either both actions are done, or neither of them. This design would cater for the correctness requirement: the number of votes in the ballot box equals the number of voters being marked 'has voted'.

At least, this is what we hoped for. Tests of the system though showed that, seemingly at haphazard moments in time, there were more votes in the ballot box than there were voters marked as 'has voted'. So the system allowed voters more than one vote.

Taking a look under the hood, a coding error was revealed in the voting process. Part of the algorithm ran as follows:

1. Identify the voter.
2. Match the voter with an entry in the register.
3. If a match is found, check that (s)he has not voted yet.

The test in the latter step had the form

```
voter.getIdentification()==identification()
```

instead of

```
equals(voter.getIdentification()==identification())
```

In other words, references were compared, rather than actual values. This is one way to win the elections.

1.5 Software Engineering Ethics

Suppose you are testing part of a big software system. You find quite a few errors and you're certainly not ready to deliver. However, your manager is pressing you. The schedule has already slipped by quite a few weeks. Your manager in turn is pressed by his boss. The customer is eagerly awaiting delivery of the system. Your manager suggests that you should deliver the system as is, continue testing, and replace the system by a better version within the next month. How would you react to this scheme? Would you simply give in? Argue with your manager? Go to his boss? Go to the customer?

The development of complex software systems involves many people: software developers, testers, technical managers, general managers, customers, etc. Within this temporary organization, the relationship between individuals is often asymmetrical: one person participating in the relationship has more knowledge about something than the other. For example, a software developer has more knowledge about the system under construction than his manager. Such an asymmetric relationship asks for trust: if the developer says that development of some component is on schedule, his manager cannot but believe this message. At least for a while. Such reliance provides opportunities for unethical behavior, such as embezzlement. This is the more so if there also is a power relationship between these individuals.

It is not surprising then that people within the software engineering community have been discussing a software engineering code of ethics. Two large organizations of professionals in our field, the IEEE Computer Society and ACM, have jointly developed such a code. The short version of this code is given in figure 1.5.

In the long version of the code, each of the principles is further refined into a set of clauses. Some of these clauses are statements of aspiration: for example, a software engineer should strive to fully understand the specifications of the software on which he works. Aspirations direct professional behavior. They require significant ethical judgment. Other clauses express obligations of professionals in general: for example, a software engineer should, like any other professional, provide service only in areas of his competence. A third type of clause is directed at specific professional behavior within software engineering: for example, a software engineer should ensure realistic estimates of the cost and schedule of any project on which he works.

There are a number of clauses which bear upon the situation of the tester mentioned above:

- Approve software only if you have a well-founded belief that it is safe, meets specifications, passes appropriate tests, and does not diminish quality of life or privacy or harm the environment (clause 1.03[1]).

- Ensure adequate testing, debugging, and review of software and related documents on which you work (clause 3.10).

[1] Clause 1.03 denotes clause no 3 of principle no 1 (Public).

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

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

Figure 1.5 Software engineering code of ethics

- As a manager, do not ask a software engineer to do anything inconsistent with this code of ethics (clause 5.11).

- Be accurate in stating the characteristics of software on which you work, avoiding not only false claims but also claims that might be supposed to be speculative, vacuous, deceptive, misleading, or doubtful (clause 6.07).

The code is not a simple algorithm to discriminate between acceptable and unacceptable behavior. Rather, the principles stated should influence you, as a software engineer, to consider who is affected by your work. The software you develop affects the public. The health, safety and welfare of the public is the primary concern of this code of ethics. Adhering to this, or a similar, code of ethics is not something to merely consider on a Friday afternoon. It should become a way of life.

The code not only addresses software engineers. It also addresses managers, in that the code indicates what might reasonably be expected from professional software engineers.

1.6 Quo Vadis?

A lot of progress has been made over the past 30 years. For each of the major phases, numerous techniques and tools have been developed. A number of these have found widespread use. In their assessment of design and coding practices for example, DeMarco and Lister found that a number of widely acclaimed techniques (such as the use of small units, strong component binding and structured programming) are indeed applied in practice and pay off (DeMarco and Lister, 1989). However, the short sketches in the preceding section (and the more elaborate discussion in the following chapters) show that a lot of research is still needed to make software engineering into a truly mature engineering discipline.

It takes some time before technology developed in research laboratories gets applied in a routine way. This holds for physical products such as the transistor, but also for methods, techniques, and tools in the area of software technology. The first version of the UNIX operating system goes right back to 1971. Only since the late 1980s, has interest in UNIX spread widely. In the early 1960s, studies of the cost of software were first made. In the 1980s there was a growing interest in quantitative models for estimating software costs (see also the later chapter on cost estimation). Dijkstra's article on programming as a human activity appeared in 1965. In the late 1970s the first introductory textbooks on structured programming were published. The term software engineering was introduced in 1968. In the 1980s large national and international programs were initiated to foster the transition of this new technology. The above list can be extended with many other examples (Redwine and Riddle, 1985). This maturation process generally takes at least 10 to 15 years.

In a seminal article entitled 'No silver bullet: essence and accidents of software engineering', Brooks (1987) discusses a number of potentially fruitful approaches to dramatically increase software productivity. His main conclusion is: there is no silver bullet. But we need not be afraid of the werewolf either. By a careful study of the many innovations and an investigation of their true merits, a lot of improvements in both quality and productivity can be achieved. The remainder of this text is devoted to a critical assessment of these technological and non-technological developments.

Several relatively recent developments have a dramatic impact on the field:

- The rise of agile methods. As noted before in this chapter, the term software engineering induces an orderly, factory-like approach to software development. This ignores the fact that for many a project, it is impossible to state the requirements upfront. They emerge as we go along. Armour (2001) compares traditional software development with shooting down a Zeppelin, and agile approaches with shooting down a supersonic plane. To shoot down a Zeppelin, we collect information on altitude, distance, velocity and the like, relay this information to the gun, aim, and shoot. This approach does not work for supersonic planes. We do not know where the intercept will be, and the missile will have to change direction while in the air. It is a challenge to try to successfully combine engineering and craft-like approaches to software development.

- There is shift from *producing* software to *using* software. Time-to-market, cost, and sheer complexity encourage organizations to assemble systems out of existing components, rather than developing those components from scratch. On one hand, builders build (pieces of) software, on the other hand integrators integrate those pieces into end-user applications. As one consequence, consumers of software often do not talk to developers anymore. Requirements come from a variety of other sources, such as helpdesk call-log analysis or market research (Sawyer, 2001). To the consumer, the software development process is not interesting any more, only the resulting product counts. This shift has given rise to new topics within software engineering, such as Component-Based Software Development (CBSD), Commercial Off-The-Shelve (COTS) components, Software Product Lines (SPL), and services.

- Software development is becoming more heterogeneous. In the old days, a software development organization had everything under control. Or so it thought. Nowadays, software is being developed by teams scattered across the globe. Part of it may be outsourced to a different organization. Software incorporates components acquired from some other supplier, or services found on the Web. As a consequence, one is not in control anymore.

To close this chapter is a list of important periodicals that contain material which is relevant to the field of software engineering:

- *Transactions on Software Engineering* (IEEE), a monthly periodical in which research results are reported;
- *Software* (IEEE), a bimonthly journal which is somewhat more general in scope;
- *Software Engineering Notes*, a bimonthly newsletter from the ACM Special Interest Group on Software Engineering;
- *Transactions on Software Engineering and Methodology* (ACM), a quarterly journal which reports research results.

- *The Journal of Systems and Software* (Elsevier), a monthly journal covering both research papers and reports of practical experiences;
- *Proceedings of the International Conference on Software Engineering* (ACM/IEEE), proceedings of the most important international conference in the field, organized every year;
- *Proceedings of the International Conference on Software Maintenance* (IEEE), organized yearly;
- *Software Maintenance and Evolution: Research and Practice* (Wiley), bimonthly journal devoted to topics in software maintenance and evolution.

1.7 Summary

Software engineering is concerned with the problems that have to do with the construction of *large* programs. When developing such programs, a phased approach is followed. First, the problem is analyzed, and then the system is designed, implemented and tested. This practice has a lot in common with the engineering of physical products. Hence the term software engineering. Software engineering, however, also differs from the engineering of physical products in some essential ways.

Software models part of the real world surrounding us, like banking or the reservation of airline seats. This world around us changes over time. So the corresponding software has to change too. It has to evolve together with the changing reality. Much of what we call software maintenance, actually is concerned with ensuring that the software keeps pace with the real world being modeled.

We thus get a process model in which we iterate earlier phases from time to time. We speak about the software life cycle.

Agile methods, reuse of components, and globalisation are some of the relatively recent trends that have a huge impact on the way we view the field. There is a shift from producing software to using software. A major consequence hereof is that a development organization looses control over what it delivers.

1.8 Further Reading

Johnson (1998) describes the early history of the software industry. The more recent state of the practice is described in (Software, 2003).

For a more elaborate discussion of the differences and similarities between software engineering and a mature engineering discipline, viz. bridge design, see (Spector and Gifford, 1986). (Leveson, 1992) compares software engineering with the development of high-pressure steam engines.

The four kinds of maintenance activities stem from (Lientz and Swanson, 1980).

The Ariane failure is described in (Jézéquel and Meyer, 1997). I found the report of the Inquiry Team at `http://www.cnes.fr/ARCHIVES/news/rapport_501.html`. An elaborate discussion of the Therac-25 accidents can be found in (Leveson and Turner, 1993). The Inquiry into the London Ambulance Service is described in (Page et al., 1993). (Neumann, 1995) is a book wholly devoted to computer-related risks. The bimonthly *ACM Software Engineering Notes* contains a column 'Risks to the public in computer systems', edited by Peter Neumann, which reports on large and small catastrophes caused by automation. (Flowers, 1996) is a collection of stories about information systems that failed, including the LAS system. (Kohno et al., 2004) and (Raba, 2004) discuss problems with one specific electronic voting system. (Petroski, 1994) is a wonderful book on failures in engineering. (Software, 1999) is a special issue with stories about successful IT projects.

The ACM/IEEE Software Engineering code of ethics is discussed in (Gotterbarn, 1999). The text of the code can also be found at `http://computer.org/tab/seprof/code.htm`. (Epstein, 1997) is a collection of (fictional) stories addressing the interaction between ethics and software engineering. (Oz, 1994) discusses ethical questions of a real-life project.

Exercises

1. Define the term software engineering.
2. What are the essential characteristics of software engineering?
3. What are the major phases in a software development project?
4. What is the difference between verification and validation?
5. Define four kinds of maintenance activity.
6. Why is the documentation of a software project important?
7. Explain the 40--20--40 rule of thumb in software engineering.
8. What is the difference between software development and software maintenance?
9. ♡ Do you think the linear model of software development is appropriate? In which cases do you think an agile approach is more appropriate? You may wish to reconsider this issue after having read the remainder of this text.
10. ♡ Discuss the major differences between software engineering and some other engineering discipline, such as bridge design or house building. Would you consider state-of-the-art software engineering as a true engineering discipline?
11. ♠ Quality and productivity are major issues in software engineering. It is

often advocated that automated tools (CASE tools) will dramatically improve both quality and productivity. Study a commercial CASE tool and assess the extent to which it improves the software development process and its outcome.

12. ♡ Medical doctors have their Hippocratic oath. Could a similar ethical commitment by software engineers be instrumental in increasing the quality of software systems?

13. ♠ Suppose you are involved in an office automation project in the printing industry. The system to be developed is meant to support the work of journal editors. The management objective for this project is to save labor cost; the editors' objective is to increase the quality of their work. Discuss possible ramifications of these opposing objectives on the project. You may come back to this question after having read chapter 9 or (Hirschheim and Klein, 1989).

14. ♡ Discuss the difference between requirements-based software development and market-driven software development (Sawyer, 2001).

15. ♡ Discuss the impact of globalisation on software development.

16. ♠ Study both the technical and user documentation of a system at your disposal. Are you satisfied with them? Discuss their possible shortcomings and give remedies to improve their quality.

17. ♠ Take a piece of software you wrote more than a year ago. Is it documented adequately? Does it have a user manual? Is the design rationale reflected in the technical documentation? Can you build an understanding of the system from its documentation that is sufficient for making non-trivial changes to it? Repeat these questions for a system written by one of your colleagues.

18. ♠ Try to gather quantitative data from your organization that reveals how much effort is spent on various kinds of maintenance activity. Are these data available at all? If so, is the pattern like that sketched in section 1.3? If not, can you explain the differences?

19. ♠ A 1999 Computer Society survey lists the following candidate fundamental principles of software engineering:

    A. Apply and use quantitative measurements in decision-making.

    B. Build with and for reuse.

    C. Control complexity with multiple perspectives and multiple levels of abstraction.

    D. Define software artifacts rigorously.

E. Establish a software process that provides flexibility.

F. Implement a disciplined approach and improve it continuously.

G. Invest in the understanding of the problem.

H. Manage quality throughout the life cycle as formally as possible.

I. Minimize software components interaction.

J. Produce software in a stepwise fashion.

K. Set quality objectives for each deliverable product.

L. Since change is inherent to software, plan for it and manage it.

M. Since tradeoffs are inherent to software engineering, make them explicit and document them.

N. To improve design, study previous solutions to similar problems.

O. Uncertainty is unavoidable in software engineering. Identify and manage it.

For each of these principles, indicate whether you (strongly) agree or (strongly) disagree, and why. You may wish to re-appraise these principles after having studied the rest of this book.