1 Introduction

We are living in an *information society* where most people are engaged in activities connected with either producing or collecting data, or organising, processing and storing data, and retrieving and disseminating stored information, or using such information for decision-making. Great developments have taken place in computer hardware technology, but the key to make this technology useful to humans lies with the software technology. In recent years software industry is exhibiting the highest growth rate throughout the world, India being no exception.

This book on software engineering is devoted to a presentation of concepts, tools and techniques used during the various phases of software development. In order to prepare a setting for the subject, in this introductory chapter, we give a historical overview of the subject of software engineering.

1.1 HISTORY OF SOFTWARE ENGINEERING

1.1.1 The Term 'Software Engineering'

While documenting the history of software engineering, we have to start with IBM 360 computer system in 1964 that combined, for the first time, the features of scientific and business applications. This computer system encouraged people to try to develop software for large and complex physical and management systems, which invariably resulted in large software systems. The need for a disciplined approach to software development was felt strongly when time and cost overruns, persisting quality problems, and high maintenance costs, etc., rose tremendously, giving rise to what was then widely termed as the "Software Crisis."

In a letter to Dr. Richard Thayer, the first editor of the IEEE Computer Society Publication on Software Engineering, Bauer (2003) who is credited to have coined the term "Software Engineering", narrates his experience of the origin of software engineering.

In the NATO Science Committee Dr. I. I. Rabi, the renowned Nobel laureate and physicist gave vent to this crisis and to the fact that the progress in software did not match the progress in hardware. The Committee set up a Study Group on Computer Science in the year 1967 with members drawn from a number of countries to assess the entire field of computer science. In its first meeting members discussed various promising scientific projects but they fell far short of a common unifying theme wanted by the Study Group. In a sudden mood of anger, Professor (Dr.) Fritz Bauer of Munich, the member from West Germany, said, “The whole trouble comes from the fact that there is so much tinkering with software. It is not made in a clean fabrication process. What we need is software engineering.” The remark shocked, but got stuck in the minds of the members of the group (Bauer 2003). On the recommendation of the Group, a Working Conference on Software Engineering was held in Garmish, West Germany, during October 7–10, 1968 with Bauer as Chairman to discuss various issues and problems surrounding the development of large software systems. Among the 50 or so participants were P. Naur, J. N. Buxton, and Dijkstra, each of whom made significant contribution to the growth of software engineering in later years.

The report on this Conference published a year later (Naur and Randell, 1969) credited Bauer to have coined the term “Software Engineering.” NATO Science Committee held its second conference at Rome, Italy in 1969 and named it the “Software Engineering Conference.”

The first International Conference on Software Engineering was held in 1973. Institute of Electronics and Electrical Engineers (IEEE) started its journal “IEEE Transactions on Software Engineering” in 1975. In 1976, IEEE Transactions on Computers celebrated its 25th anniversary. To that special issue, Boehm contributed his now-famous paper entitled, Software Engineering (Boehm 1976), that clearly defined the scope of software engineering.

In 1975, Brooks (1975), who directed the development of IBM 360 operating system software over a period of ten years involving more than 100 man-months wrote his epoch-making book, “The Mythical Man-Month” where he brought out many problems associated with the development of large software programs in a multi-person environment.

In 1981, Boehm (1981) brought out his outstanding book entitled “Software Engineering Economics” where many managerial issues including the time and cost estimate of software development were highlighted.

Slowly and steadily software engineering grew into a discipline that not only recommended technical but also managerial solutions to various issues of software development.

1.1.2 Development of Tools and Techniques of Software Engineering

Seventies saw the development of a wide variety of engineering concepts, tools, and techniques that provided the foundation for the growth of the field. Royce (1970) introduced the phases of the software development life cycle. Wirth (1971) suggested stepwise refinement as method of program development. Hoare *et al.* (1972) gave the concepts of structured programming and stressed the need for doing away with GOTO statements. Parnas (1972) highlighted the virtues of modules and gave their specifications.

Endres (1975) made an analysis of errors and their causes in computer programs. Fagan (1976) forwarded a formal method of code inspection to reduce programming errors. McCabe (1976) developed flow graph representation of computer programs and their complexity measures that helped in testing. Halstead (1977) introduced a new term “Software Science” where he gave novel ideas for using information on number of unique operators and operands in a program to estimate its size and complexity. Gilb (1977) wrote the first book on software metrics. Jones (1978) highlighted misconceptions surrounding software quality and productivity and suggested various quality and productivity measures. DeMarco (1978) introduced the concept of data flow diagrams for structured analysis. Constantine and Yourdon (1979) gave the principles of structured design.

Eighties saw the consolidation of the ideas on software engineering. Boehm (1981) presented the COCOMO model for software estimation. Albrecht and Gaffney (1983) formalised the concepts of "function point" as a measure of software size. Ideas proliferated during this decade in areas such as process models, tools for analysis, design and testing. New concepts surfaced in the areas of measurement, reliability, estimation, reusability and project management.

This decade witnessed also the publication of an important book entitled, "Managing the Software Process" by Humprey (1989), where the foundation of the capability maturity models was laid.

Nineties saw a plethora of activities in the area of software quality, in particular, in the area of quality systems. Paulk *et al.* (1993) and Paulk (1995) developed the capability maturity model. Gamma *et al.* (1995) gave the concepts of "design patterns." This decade also saw publications of many good text books on software engineering (Pressman 1992, Sommerville 1996). This decade has also seen the introduction of many new ideas such as software architecture (Shaw and Garlan, 1996) and component-based software engineering (Pree 1997). Another development in this decade is the object-oriented analysis and design and unified modeling language (Rumbaugh *et al.* 1998 and Booch *et al.* 1999).

The initial years of the twenty-first century have seen the consolidation of the field of design patterns, software architecture, and component-based software engineering.

We have stated above that the many problems encountered in developing large software systems were bundled into the term software crisis and the principal reason for founding the discipline of software engineering was to defuse the software crisis. In the next section we shall see more clearly the factors that constituted the software crisis.

1.2 SOFTWARE CRISIS

During the late 1960s and 1970s, there was an outcry over an impending "software crisis." The symptoms of such a crisis surfaced then and are present even today. The symptoms are the following:

1. Software cost has shown a rising trend, outstripping the hardware cost. Boehm (1976, 1981) indicated that since the fifties, the percentage of total cost of computation attributable to hardware has dramatically reduced and that attributable to software has correspondingly increased (Fig. 1.1). Whereas software cost was only a little over 20% in the 1950's, it was nearly 60% in the 1970's, and about 80% in the 1980's. Today, the computer system that we buy as 'hardware' has generally cost the vendor about three times as much for the software as it has for the hardware (Pressman 1992).

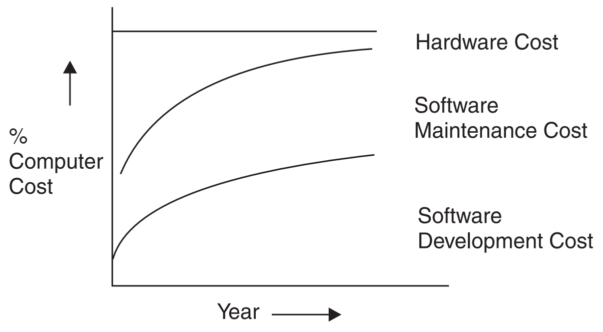

Fig. 1.1. Hardware and software costs

2. Software maintenance cost has been rising and has surpassed the development cost. Boehm (1981) has shown that the bulk of the software cost is due to its maintenance rather than its development (Fig. 1.1).
3. Software is almost always delivered late and exceeds the budgeted cost, indicating time and cost overruns.
4. It lacks transparency and is difficult to maintain.
5. Software quality is often less than adequate.
6. It often does not satisfy the customers.
7. Productivity of software people has not kept pace with the demand of their services.
8. Progress on software development is difficult to measure.
9. Very little real-world data is available on the software development process. Therefore, it has not been possible to set realistic standards.
10. How the persons work during the software development has not been properly understood.

One of the earliest works that explained to a great extent the causes of software crisis is by Brooks (1972). We shall get in the next section a glimpse of the work of Brooks.

1.3 EVOLUTION OF A PROGRAMMING SYSTEM PRODUCT

In his book 'The Mythical Man-Month' Brooks (1975) narrates his experience on the development of the IBM 360 operating system software. Among his many significant observations, one that is relevant at this stage is his observation on the effect of multiple users and multiple developers on the software development time. He distinguishes a program written by a person for his (her) use from a programming product, a programming system, and from a programming systems product.

A *program* is complete in itself, run by the author himself (herself), and is run on the machine on which it is developed. A *programming product* is a program that is written in a generalised fashion such that it can be run, tested, repaired, and extended by anybody. It means that the program must be tested, range and form of input explored, and these are well-recorded through documentation. A program, when converted into a programming product, costs, as a rule of thumb, three times as much as itself.

A *programming system* is a collection of interacting programs, coordinated in function and disciplined in format, so that the assemblage constitutes an entire facility for large tasks. In a programming system component, inputs and outputs must conform in syntax and semantics with precisely defined interfaces, use a prescribed budget of resources—memory space, input-output devices, and computer time, and must be tested with other components in all expected combinations. It generally costs at least three times as much as a stand-alone program of the same function.

A *programming system product* has all the features of a programming product and of a programming system. It generally costs at least nine times as much as a stand-alone program of the same function.

Figure 1.2 shows the evolution of a programming system product. It shows how product cost rises as a program is slowly converted into a programming system product. This discussion by Brooks is meant to bring home the point that developing software containing a set of interacting programs for the use by persons other than the developers requires much more time and effort than those required for developing a program for use by the developer. Since most software today is used by persons other than the developers, the cost of software development is surely going to be prohibitive. Software engineering methods, tools, and procedures help in streamlining the development activity so that the software is developed with high quality and productivity and with low cost.

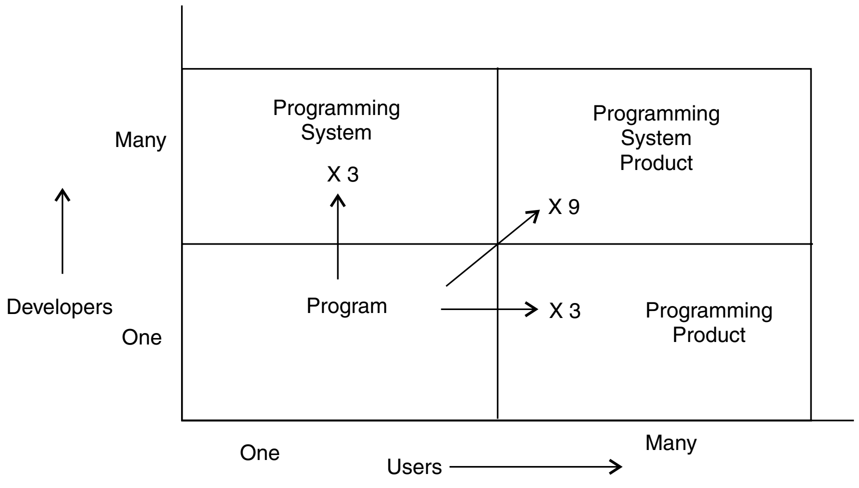

Fig. 1.2. Levels of programming

Some of the major reasons for this multiplying effect of multiple users and developers on software development time and, in general, the genesis of the software crisis can be better appreciated if we understand the characteristics of software and the ways they are different from those in the manufacturing environment.

1.4 CHARACTERISTICS OF SOFTWARE

Software is a logical rather than a physical system element. Therefore, software has characteristics that are considerably different from those of hardware (Wolverton 1974, and Pressman 1992). Some of the major differences are the following:

1. Software is developed or engineered, it is not manufactured.
   - The concept of 'raw material' is non-existent here. It is better visualised as a process, rather than a product (Jensen and Tonies, 1979).
   - The 'human element' is extremely high in software development, compared to manufacturing.
   - The development productivity is highly uncertain, even with standard products, varying greatly with skill of the developers.
   - The development tools, techniques, standards, and procedures vary widely across and within an organisation.

- Quality problems in software development are very different from those in manufacturing. Whereas the manufacturing quality characteristics can be objectively specified and easily measured, those in the software engineering environment are rather elusive.

2. Software development presents a job-shop environment.
   - Here each product is custom-built and hence unique.
   - It cannot be assembled from existing components.
   - All the complexities of a job shop (*viz.*, the problems of design, estimating, and scheduling) are present here.
   - Human skill, the most important element in a job shop, is also the most important element in software development.
3. Time and effort for software development are hard to estimate.
   - Interesting work gets done at the expense of dull work, and documentation, being a dull work, gets the least priority.
   - Doing the job in a clever way tends to be a more important consideration than getting it done adequately, on time, and at reasonable cost.
   - Programmers tend to be optimistic, not realistic, and their time estimates for task completion reflect this tendency.
   - Programmers have trouble communicating.
4. User requirements are often not conceived well enough; therefore a piece of software undergoes many modifications before it is implemented satisfactorily.
5. There are virtually no objective standards or measures by which to evaluate the progress of software development.
6. Testing a software is extremely difficult, because even a modest-sized program (< 5,000 executable statements) can contain enough executable paths (*i.e.*, ways to get from the beginning of the program to the end) so that the process of testing each path though the program can be prohibitively expensive.
7. Software does not wear out.
   - Software normally does not lose its functionality with use.
   - It may lose its functionality in time, however, as the user requirements change.
   - When defects are encountered, they are removed by rewriting the relevant code, not by replacing it with available code. That means that the concept of replacing the defective code by spare code is very unusual in software development.
   - When defects are removed, there is likelihood that new defects are introduced.
8. Hardware has physical models to use in evaluating design decisions. Software design evaluation, on the other hand, rests on judgment and intuition.
9. Hardware, because of its physical limitations, has practical bound on complexity because every hardware design must be realised as a physical implementation. Software, on the other hand, can be highly complex while still conforming to almost any set of needs.

10. There are major differences between the management of hardware and software projects. Traditional controls for hardware projects may be counterproductive in software projects. For example, reporting percent completed in terms of Lines of Code can be highly misleading.

    It is now time to give a few definitions. The next section does this.

1.5 DEFINITIONS

Software

According to Webster's New Intercollegiate Dictionary, 1979,

"Software is the entire set of programs, procedures and related documentation associated with a system and especially a computer system."

The New Webster's Dictionary, 1981, reworded the definition, orienting it completely to computers:

"Software is the programs and programming support necessary to put a computer through its assigned tasks, as distinguished from the actual machine."

A more restrictive but functional definition is given by Blum (1992):

"Software is the detailed instructions that control the operation of a computer system. Its functions are to (1) manage the computer resources of the organisation, (2) provide tools for human beings to take advantage of these resources, and (3) act as an intermediary between organisations and stored information."

Gilb (1977) defines two principal components of software:

1. *Logicware*, the logical sequence of active instructions controlling the execution sequence (sequence of processing of the data) done by the hardware, and
2. *Dataware*, the physical form in which all (passive) information, including logicware, appears to the hardware, and which is processed as a result of the logic of the logicware.

Figure 1.3 (Gilb 1977) shows not only these two elements of a software system, but it also shows the other components as well.

There are eight levels of software that separate a user form the hardware. Following Gilb (1977) and Blum (1992), we show these levels in Fig. 1.4.

A. Hardware Logic	1. Machine Micrologic
B. System Software	2. Supervisor or Executive
	3. Operating System
	4. Language Translators
	5. Utility Programs

C. Application Software	6. Inquiry, File, and Database Software
	7. Programming and Assembly Languages and Programs
D. End-user Software	8. Fourth-Generation Languages and User Programs, such as SPSS, dbase-IV, and Lotus 1-2-3, SQL, etc.

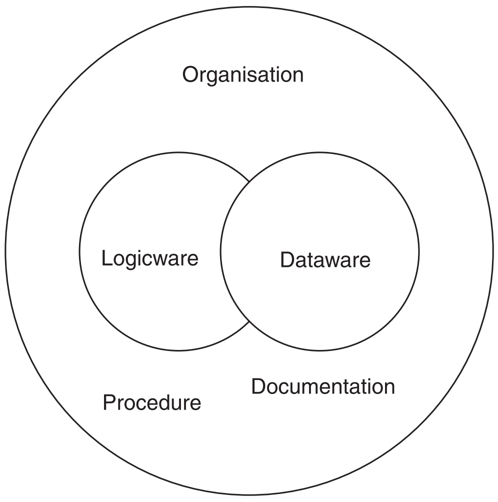

Fig. 1.3. Components of software systems

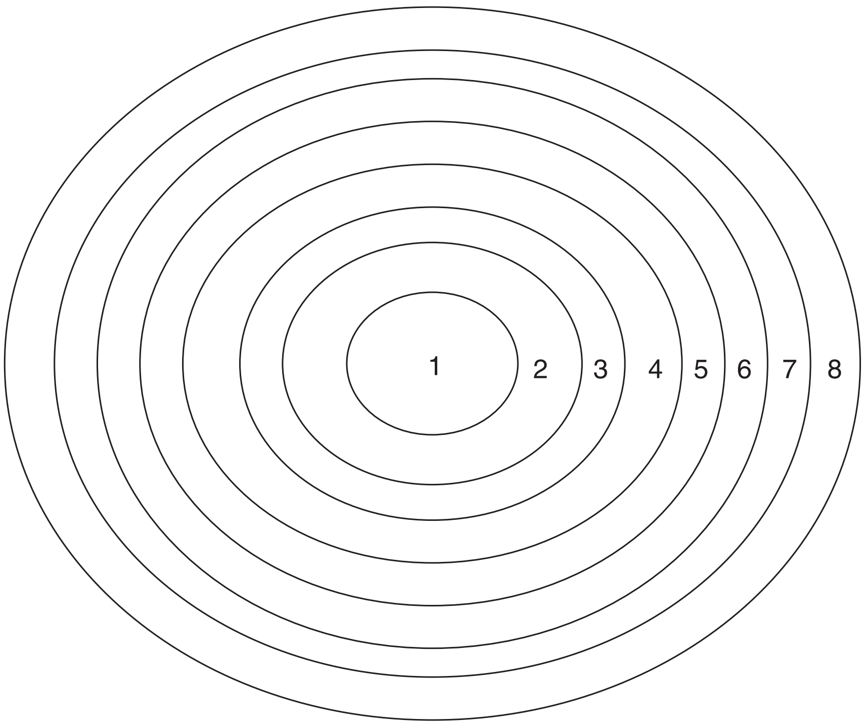

Fig. 1.4. Levels of software

What it is important to note here is that, contrary to popular belief, software includes not only the *programs* but also the *procedures* and the *related documentation*. Also important to note is that the word *software* is a collective noun just as the word *information* is; so the letter *s* should not be used after it. While referring to a number of packages, one should use the term *software packages*. Similarly, one should use the terms *software products*, *pieces of software*, and so on, and not the word *softwares*.

Engineering

New Intercollegiate Webster's Dictionary, 1979, defines the term *engineering* as

"the application of science and mathematics by which the properties of matter and the sources of energy in nature are made useful to man in structures, machines, products, systems and processes."

Thus, engineering denotes the application of scientific knowledge for practical *problem solving*.

Software Engineering

Naur (Naur and Randell 1969) who co-edited the report on the famous NATO conference at Garnish also co-authored one of the earliest books on the subject (Naur *et al.* 1976). In this book, the ideas behind *software engineering* were given as the following:

- Developing large software products is far more complex than developing stand-alone programs.
- The principles of engineering design should be applied to the task of developing large software products.

There are as many definitions of "Software Engineering" as there are authors. We attempt to glimpse through a sample of definitions given by exponents in the field.

Bauer (1972) gave the earliest definition for software engineering (Bauer 1972, p. 530):

"… the establishment and use of sound engineering principles (methods) in order to obtain economically software that is reliable and works on real machines."

According to Boehm (1976), software engineering is

"… the practical application of scientific knowledge in the design and construction of computer programs and the associated documentation required to develop, operate and maintain them."

Boehm (1976) expanded his idea by emphasising that the most pressing software development problems are in the area of requirements analysis, design, test, and maintenance of application software by technicians in an *economics-driven context* rather than in the area of detailed design and coding of system software by experts in a relatively *economics-independent* context.

DeRemer and Kron (1976) recognise *software engineering* to deal with *programming-in-the-large*, while Parnas (1978) is of the view that *software engineering* deals with '*multi-person construction of multi-version software*.'

Sommerville (1992) summarises the common factors involving software engineering:

1. Software systems are built by teams rather than individuals.
2. It uses engineering principles in the development of these systems that include both technical and non-technical aspects.

A more recent definition by Wang and King (2000) considers software engineering as a discipline and makes the engineering principles and product attributes more explicit:

"Software engineering is a discipline that adopts engineering approaches such as established methodologies, process, tools, standards, organisation methods, management methods, quality assurance systems, and the like to develop large-scale software with high productivity, low cost, controllable quality, and measurement development schedules."

Conventional Engineering and Software Engineering: Similarities and Differences

It is obvious from some of the above-stated definitions that software engineering shares quite a few things common with the principles of conventional engineering. Here we outline these similarities and a few differences between the two disciplines.

Jensen and Tonies (1979) consider software engineering to be related to the design of software or data processing products and to belong to its *problem solving* domain, encompassing the class of problems related to software and data processing. They expand their idea by drawing analogy from the methods that are generally used in engineering. According to them, just as the celebrated *scientific method* is used in the field of scientific research, the steps of *engineering design process* are used in the process of problem solving in the field of engineering. These steps, which are mostly iterative, are: (1) Problem formulation, (2) Problem analysis, (3) Search for alternatives, (4) Decision, (5) Specification, and (6) Implementation. Jenson and Tonies suggest that these steps are applicable to the field of software engineering as well.

Pressman (1992) considers software engineering as an outgrowth of hardware and systems engineering, encompassing a set of three key elements—methods, tools and procedures which enable the manager to control the process of software development. According to Pressman, *methods* provide the technical "how to's" for building software; *tools* provide automated or semi-automated support for methods; and *procedures* define the sequence of applying the methods, the deliverables, the controls, and the milestones.

Wang and King (2000) have highlighted the philosophical foundations of software engineering. Compared to traditional engineering disciplines, software engineering shows a few remarkable differences:

- In conventional engineering, one moves from an abstract design to a concrete product. In contrast, in software engineering, one moves from design to coding (that can be considered as abstract).

Software Engineering:	Abstract Design	$\longrightarrow$	More Abstract Code
Manufacturing Engineering:	Abstract Design	$\longrightarrow$	Concrete Products

- The problem domains of software engineering can be almost anything, from word processing to real-time control and from games to robotics. Compared to any other engineering discipline, it is thus much wider in scope and thus offers greater challenges.

- Traditional manufacturing engineering that normally emphasises mass production is loaded with production features. Thus, it is highly *production intensive*. Software engineering, on the other hand, is inherently *design intensive*.
- *Product standardisation* helps in cost reduction in manufacturing, whereas such a possibility is remote in software engineering. The possibility of *process standardisation*, however, is very high in the latter.
- Unlimited number of domain- and application-specific notions prevails in engineering disciplines. Software engineering, on the other hand, uses a limited, but universal, number of concepts, for example, standard logical structures of sequence, condition, and repetition.

1.6 NO SILVER BULLETS

In a widely-referred paper, Brooks, Jr. (1986) draws analogy of software projects with werewolves in the American folklore. Just as the werewolves transform unexpectedly from the familiar into horrors and require bullets made of silver to magically lay them to rest, the software projects, appearing simple and without problem, can transform into error-prone projects with high time and cost overruns. There is no silver bullet to ameliorate this problem, however.

According to Brooks, the *essence* of difficulties associated with software engineering lies with specification, design, and testing of the conceptual constructs while the error during representation are *accidents*. Software engineering must address the *essence*, and not the *accidents*.

The properties of essence of modern software systems, according to Brooks, Jr. (1986) are the following:

1. Complexity:	No two parts of a software product are alike.
2. Conformity:	Unlike natural laws in the physical systems, there does not seem to be a unifying theory for software systems.
3. Changeability:	While manufactured products do not change very frequently, software products change, particularly with user requirements changing.
4. Invisibility:	No really geometric representation, unlike a plan for a building or a drawing of the design of a machine, can represent the design of a software program.

Brooks, Jr. is of the opinion that the past breakthroughs, like high-level languages, time-sharing facility, and unifying programming environments (such as Unix), have attacked only the accidental problems of software engineering, not the essential ones. He is also skeptical about the ability of such developments as advances in other high-level languages, object-oriented programming, artificial intelligence, expert systems, automatic programming, program verification, programming environments and tools, and workstations in solving the essential problems of software engineering.

Brooks, Jr. suggests that the following developments have high potential in addressing the essential problems of software engineering:

1. *Buy rather than build.* Tested components already developed and in use are the best candidates to be reused in new software products. They will be error free. However, the

components have to be selected and they have to be properly integrated with the new software being developed.

2. *Requirements refinement and rapid prototyping*. Prototyping is a very useful method to elicit user information requirement. It helps to find out core requirements which are then refined when new prototypes are displayed to the users.
3. *Incremental development.* Developing the core functional requirements and then incrementally adding other functions hold the key to developing error-free software products.
4. *Creative designers.* The software firms should retain the best and the most skilled designers because they hold the key to bring out quality software products.

We end this chapter by stating a few myths surrounding development of software systems.

1.7 SOFTWARE MYTHS

Pressman (1992) has compiled the following myths that prevail in the software industry:

A. Management Myths:

- We already have a book that's full of standards and procedures for building software. Won't that provide my people with everything they need to know?
- My people do have state-of-the-art software development tools; after all, we buy them the newest computers.
- If we get behind schedule, we can add more men and catch up.

B. Customer's Myths:

- A general statement of objectives is sufficient to begin writing programs—we can fill in the details later.
- Project requirements continually change, but change can be easily accommodated because software is flexible.

C. Practitioner's Myths:

- Once we write the program and get it to work, our job is done.
- Until I get the program "running," I really have no way of assessing its quality.
- The only deliverable for a successful project is the working program.

As software engineering tools and techniques are developed and practiced, these myths have given way to genuine concern for new development tools and to a strong desire to know them. The following chapters elucidate them with examples and with reference to their development from the past to the present.

REFERENCES

Albrecht A. J. and J. E. Gaffney (1983), Software Function, Lines of Code and Development Effort Prediction: A Software Science Validation, IEEE Transactions on Software Engineering, vol. 9, no. 6, pp. 639–647.

Bauer, F. L. (1972), Software Engineering, Information Processing 71, North-Holland Publishing Co., Amsterdam.

Bauer, F. L. (1976), Software Engineering, in Ralston, A. and Mek, C. L. (eds.), Encyclopaedia of Computer Science, Petrocelli/charter, New York.

Bauer, F. L. (2003), The Origin of Software Engineering—Letter to Dr. Richard Thayer in Software Engineering, by Thayer, R. H. and M. Dorfman (eds.) (2003), pp. 7–8, John Wiley & Sons, Inc., N.J.

Blum, B. I. (1992), Software Engineering: A Holistic View, Oxford University Press, New York.

Boehm, B. W. (1976), Software Engineering, IEEE Transactions on Computers, vol. 25, no. 12, pp. 1226–1241.

Boehm B. W. (1981), Software Engineering Economics, Englewood Cliffs, NJ: Prentice Hall, Inc.

Booch, G., J. Rumbaugh, and I. Jacobson (1999), The Unified Modeling Language User Guide, Addison-Wesley Longman, Singapore Pte. Ltd.

Brooks, F. (1975), The Mythical Man-Month, Reading, MA: Addison-Wesley Publishing Co.

Brooks, F. P., Jr. (1986), No Silver Bullet: Essence and Accidents of Software Engineering, Information Processing '86, H. J. Kugler (ed.), Elsevier Science Publishers, North Holland, IFIP 1986.

DeMarco. T. (1978), Structured Analysis and System Specification, Yourdon Press, New York.

DeRemer, F. and H. Kron, (1976), Programming-in-the-Large versus Programming-in-the-Small, IEEE Transactions on Software Engineering, vol. 2, no. 2, pp. 80–86.

Endres, A. (1975), An Analysis of Errors and Their Causes in System Programs, IEEE Transactions on Software Engineering, vol. 1, no. 2, pp. 140–149.

Fagan, M. E. (1976), Design and Code Inspections to Reduce Errors in Program Development, IBM Systems J., vol. 15, no. 3, pp. 182–211.

Gamma, E., R. Helm, R. Johnson, and J. Vlissides (1995), Design Patterns: Elements of Reusable Object-Oriented Software, MA: Addison-Wesley Publishing Company, International Student Edition.

Ghezzi C., M. Jazayeri, and D. Mandrioli (1994), Fundamentals of Software Engineering, Prentice-Hall of India Private Limited, New Delhi.

Gilb, T. (1977), Software Metrics, Cambridge, Mass: Winthrop Publishers, Inc.

Halstead, M. H. (1977), Elements of Software Science, North Holland, Amsterdam.

Hoare, C. A. R., E-W, Dijkstra, and O-J. Dahl (1972), Structured Programming, Academic Press, New York.

Humphrey, W.S. (1989), Managing the Software Process, Reading MA: Addison-Wesley.

Jensen, R. W. and C. C. Tonies (1979), Software Engineering, Englewood Cliffs, NJ: Prentice Hall, Inc.

Jones, T. C. (1978), Measuring Programming Quality and Productivity, IBM Systems J., vol. 17, no. 1, pp. 39–63.

McCabe, T. J. (1976), A Complexity Measure, IEEE Transactions on Software Engineering, vol. 2, no. 4, pp. 308–320.

McDermid, J. A., ed. (1991), Software Engineering Study Book, Butterworth-Heinemann Ltd., Oxford, UK.

Naur, P. and Randell, B. (eds.) (1969), Software Engineering: A Report on a Conference Sponsored by the NATO Science Committee, NATO.

Naur, P., B. Randell, and J. Buxton (eds.) (1976), Software Engineering: Concepts and Techniques, Petrocelli/Charter, New York.

Parnas, D. L. (1972), A Technique for Module Specification with Examples, Communications of the ACM, vol. 15, no. 5, pp. 330–336.

Parnas, D. L. (1978), Some Software Engineering Principles, in Structured Analysis and Design, State of the Art Report, INFOTECH International, pp. 237–247.

Paulk, M. C. Curtis, B., Chrissis, M. B. and Weber, C. V. (1993), Capability Maturity Model, Version 1-1, IEEE Software, vol. 10, no. 4, pp. 18–27.

Paulk, M. C. (1995), How ISO 9001 Compares with the CMM, IEEE Software, January, pp. 74–83.

Pree, W. (1997), Component-Based Software Development—A New Paradigm in Software Engineering, Software Engineering Conference, ASPEC 1997 and ICSC 1997, Proceedings of Software Engineering Conference 1997, 2–5 December 1997, pp. 523-524.

Pressman, R. S. (1992), Software Engineering: A Practitioner's Approach, McGraw-Hill International Editions, Third Edition, Singapore.

Royce, W. W. (1970), Managing of the Development of Large Software Systems, in Proceedings of WESTCON, San Francisco, CA.

Rumbaugh, J., Jacobson, I., and Booch, G. (1998), The Unified Modeling Language Reference Manual, ACM Press, New York.

Shaw, M. and D. Garlan (1996), Software Architecture: Perspectives on an Emerging Discipline, Prentice-Hall.

Sommerville, I. (1996), Software Engineering (Fifth edition), Addison-Wesley, Reading MA.

Wang, Y. and G. King (2000), Software Engineering Process: Principles and Applications, CRC Press, New York.

Wang, Y., Bryant, A., and Wickberg, H. (1998), A Perspective on Education of the Foundations of Software Engineering, Proceedings of 1st International Software Engineering Education Symposium (SEE'98), Scientific Publishers OWN, Poznars, pp. 194–204.

Wirth, N. (1971), Program Development by Stepwise Refinement, Communications of the ACM, vol. 14, no. 4, pp. 221–227.

Wolverton, R. W. (1974), The Cost of Developing Large-scale Software, IEEE Transactions on Computers, June, pp. 282–303.