CHAPTER 1

INTRODUCTION

Computers are commercially being used for over the past sixty years. If we examine the way computers have evolved over this period, we can see that in the early days computers were very slow and lacked sophistication. The computational power and sophistication of computers have increased ever since, while their prices have dropped dramatically. The improvements to their speed and reductions to their cost were brought about by several technological breakthroughs that occurred at regular intervals.

The more powerful a computer is, the more sophisticated programs it can run. Therefore, with the every increase in capabilities of computers, software engineers have been called upon to solve larger and more complex problems, and that too in cost-effective and efficient ways. Software engineers have admirably coped with this challenge by innovating and by building upon their past programming experience. All these innovations and experiences have given rise to the discipline of software engineering. Let us now examine the scope of the software engineering discipline more closely.

What is software engineering? The essence of all past programming experiences and innovations for writing good quality programs in cost-effective and efficient ways have been systematically organized into a body of knowledge. This knowledge forms the foundation of the software engineering principles. From this point of view, we can define the scope of software engineering as follows.

> Software engineering discusses systematic and cost-effective techniques to software development. These techniques have resulted from innovations as well as lessons learnt from past mistakes. Alternatively, we can view software engineering as the engineering approach to develop software.

But, what exactly is an engineering approach to develop software? Let us try to answer this question using an analogy. Suppose you have asked a petty contractor to build a small house for you. Petty contractors are not really experts in house building. They normally carry out minor repair works only and may at most undertake very small building work such as the construction of boundary walls, etc. Now faced with the task of building a complete house, your petty contractor would draw upon all the knowledge he has regarding house building. Yet, he would often be left with no clue regarding what to do. For example, he might not know the optimal proportion in which cement and sand should be mixed to realize sufficient strength for supporting the roof. In such situations, he would have to fall back upon his intuitions. He would normally succeed in his work, if the house you asked him to construct is sufficiently small. Of course, the house constructed by him may not look as good as one constructed by a professional, may lack proper planning, and display several defects and imperfections. It may even cost more and take longer to build.

Now, suppose you entrust your petty contractor to build a large 50-storeyed commercial complex. He might exercise prudence, and politely refuse to undertake your request. On the other hand, he might be ambitious and agree to undertake the task. In the latter case, he is sure to fail. The failure might come in several forms—the building might collapse during the construction stage itself due to his ignorance of the theories concerning the strengths of materials; the construction might get unduly delayed, since he may not prepare proper estimations and detailed plans regarding the types and quantities of raw materials required, the times at which these are required, etc. In short, to be successful in constructing a building of large magnitude, one needs a thorough understanding of civil and architectural engineering techniques such as analysis, estimation, prototyping, planning, designing and testing, etc. Similar things happen in case of software development projects as well. For sufficiently small-sized problems, one might proceed according to one's intuition and succeed; though the solution may have several imperfections, cost more, take longer to complete, etc. But, failure is almost certain, if one undertakes a large-scale software development work without a sound understanding of the software engineering principles.

Why is software engineering neither a form of science nor an art? Let us now analyze why software engineering discipline was not classified either as a form of science or an art. There exist several fundamental issues that set software engineering (and other engineering disciplines such as civil engineering) apart, from both science and art. Some of these important issues are the following:

- Engineering disciplines such as software engineering make heavy use of past experience. The past experiences are systematically arranged and theoretical basis for them are provided wherever possible. Whenever no reasonable theoretical justification could be provided, the past experiences are adopted as rules of thumb. In contrast to the approach adopted by engineering disciplines, only exact solutions are accepted by science and that too when backed by rigorous proofs.
- In engineering disciplines, while designing a system, several conflicting goals might have to be optimized. In such situations, no unique solution may exist and several alternate solutions may be proposed. While selecting an appropriate solution out of several candidate solutions, various trade-offs are made based on issues such as cost, maintainability and usability. Therefore, to arrive at the final solution, several iterations and backtracking may have to be performed. In science, on the other hand, only unique solutions are possible.
- In an engineering discipline, a pragmatic approach to cost-effectiveness is adopted and economic concerns are addressed. Science normally does not concern itself with practical issues such as cost, maintainability and usability implications of a solution.
- Engineering disciplines are based on well-understood and quantitative principles. Art, on the other hand, is often based on subjective judgement which are based on qualitative attributes.

1.1 THE SOFTWARE ENGINEERING DISCIPLINE—ITS EVOLUTION AND IMPACT

In this section, we first briefly review how the software engineering discipline has evolved to its present form, starting with a humble beginning about six decades ago. We then point out that in spite of various shortcomings of the software engineering principles, they are still the best bet against the present software crisis.

1.1.1 Evolution of an Art into an Engineering Discipline

Software engineering principles have evolved over the last sixty years with contributions from numerous researchers and software professionals. To get a feel of these contributions and how they have shaped the evolution of the software engineering discipline, let us recount a few glimpses of the past.

The early programmers used an exploratory also called build and fix programming style.

In the build and fix (exploratory) style, normally a poor quality program is quickly developed without making any specification, plan, or design. The different imperfections that are subsequently noticed while using or testing are fixed.

The exploratory programming style is a very informal style of program development approach, and there are no set rules or recommendations that one has to adhere to— every programmer himself evolves his own software development techniques solely guided by his intuition, experience, whims and fancies. The exploratory style is also the one that is normally adopted by all students and novice programmers who do not have any exposure to software engineering principles. (We shall subsequently see that such a programming style results in poor quality and unmaintainable code and also makes program development very expensive and time-consuming.) We can consider the exploratory program development style as an art–since, any art is mostly guided by the intuition.

There are many stories about programmers in the past who were like proficient artists and could write good programs based on some esoteric knowledge. In contrast, programmers in modern software industry rarely make use of such esoteric knowledge, and rather develop software by applying some well-understood principles. If we analyze the evolution of software development styles over the last fifty years, we can easily notice that it has evolved from an esoteric art form to a craft form, and then has slowly emerged as an engineering discipline. As a matter of fact, this growth pattern is not very different from that seen in other engineering disciplines.

Irrespective of whether it is iron making, paper making, software development or building construction; evolution of technology has followed strikingly similar patterns. A schematic representation of this pattern of technology development is shown in Figure 1.1. It can be seen from Figure 1.1 that every technology initially starts as a form of art. Over time, it graduates to a craft, and finally emerges as an engineering discipline. Let us illustrate this fact using an example. Consider the evolution of the iron making technology. In ancient times, only a few people knew how to make iron. Those who knew iron making, kept it a closely-guarded secret. This esoteric knowledge got transferred from generation to generation as a family secret. Slowly, technology graduated from an art to a craft form where tradesmen shared their knowledge with their apprentices, and the knowledge pool continued to grow. Much later, through a systematic organization of knowledge, and incorporation of scientific basis, modern steel making technology emerged. The story of the evolution of the software engineering discipline is not much different. In the early days of programming, there were good programmers and bad programmers. The good programmers knew certain principles (or *tricks*) that they seldom shared with the bad programmers. Over the years, all such good principles (or *tricks*) along with research innovations have systematically been organized into a body of knowledge that forms the discipline of software engineering.

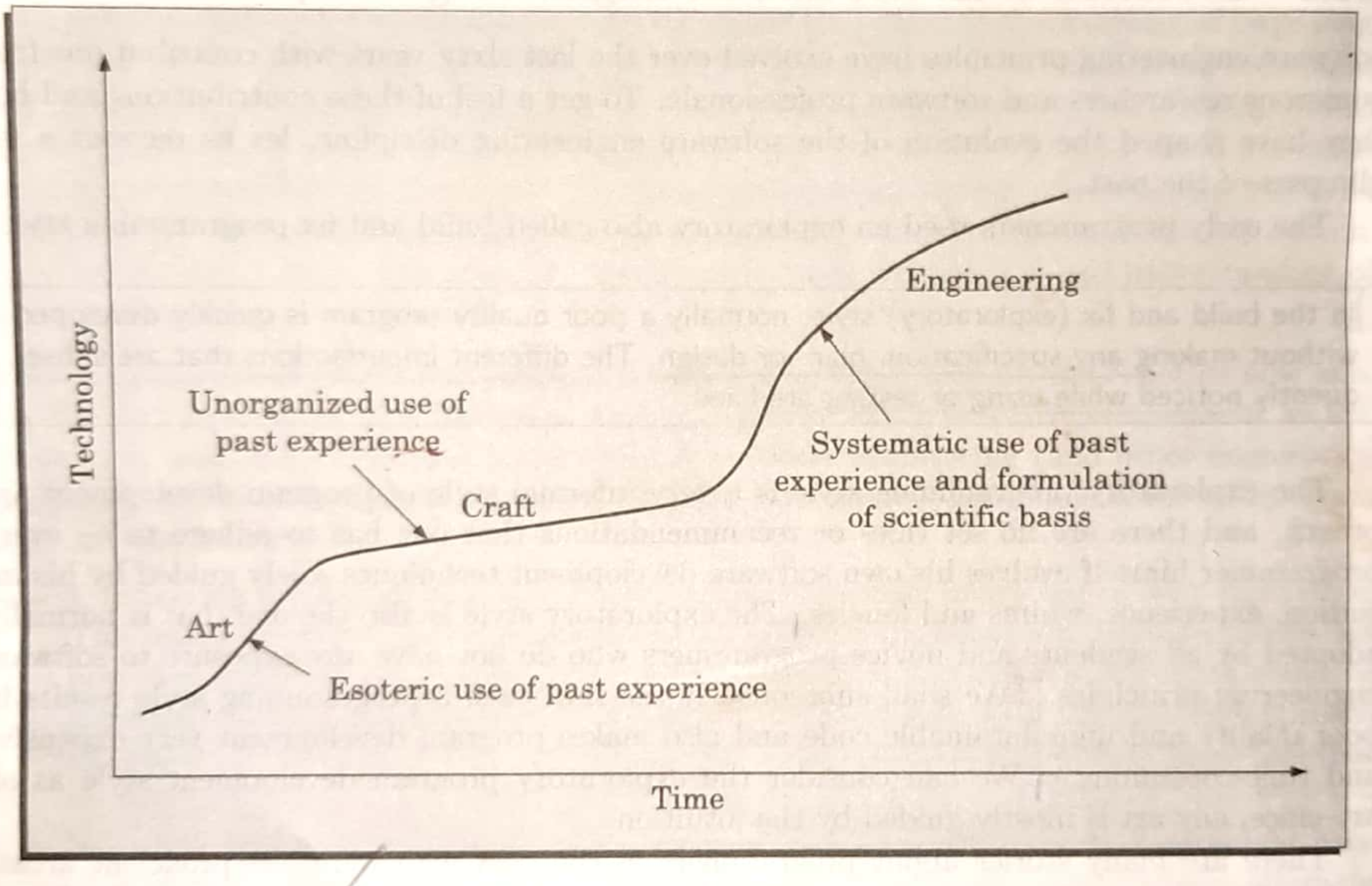
Technology
Engineering
Unorganized use of past experience
Craft
Systematic use of past experience and formulation of scientific basis
Art
Esoteric use of past experience
Time

Figure 1.1: Evolution of technology with time.

Software engineering principles are now being widely used in industries, and new principles are still continuing to emerge at a very rapid rate making this discipline highly dynamic. In spite of its wide acceptance, critics point out that many of the methodologies and guidelines provided by the software engineering discipline lack scientific basis, are subjective, and are often inadequate. Yet, there is no denying the fact that adopting software engineering techniques facilitates development of high quality software in a cost-effective and timely manner. Software engineering practices have proven to be indispensable to the development of large software products—though exploratory styles can often be used successfully to develop small programs.

1.3.3 Why Study Software Engineering?

Let us examine the skills that can be acquired from a study of the software engineering principles. The following two are possibly the most important skills you could be acquiring after completing a study of software engineering:

1. The skill to participate in development of large software products. You can meaningfully participate in a team effort to develop a large software product only after learning the systematic techniques that are being used in the industry.
2. You would learn how to effectively handle complexity in a software development problem. In particular, you would learn how to apply the principles of abstraction and decomposition to handle complexity during software specification, design, construction and testing.

Besides the above two important skills, you would also be learning the techniques of software requirements specification user interface development, quality assurance, testing, project management, maintenance, etc.

As we had already mentioned, small programs can also be written without using software engineering principles. However, even if you intend to write small programs, the software engineering principles could help you to achieve higher productivity, and at the same time enable you to produce better quality programs.