**UNIT-I**

**CHAPTER-1**

**INTRODUCTION TO SOFTWARE ENGINEERING**

**Defining Software**

Software is defined as

1. Instructions: Programs that when executed provide desired function, features, and performance

2. Data structures: Enable the programs to adequately manipulate information

3. Documents: Descriptive information in both hard copy and virtual forms that describes the operation and use of the programs.

**Software** **Engineering**

The term is made of two words, software and engineering.

**Software** is more than just a program code. A program is an executable code, which serves some computational purpose. Software is considered to be collection of executable programming code, associated libraries and documentations. Software, when made for a specific requirement is called software product.

**Engineering** on the other hand, is all about developing products, using well-defined, scientific principles and methods.

**Software engineering** is an engineering branch associated with development of software product using well defined scientific principles, methods and procedures. The outcome of software engineering is an efficient and reliable software product.

**IEEE defines software engineering as**:

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

**What is software engineering?**

Software engineering discusses systematic and cost-effective techniques for software development. These techniques help develop software using an engineering approach.

**EVOLUTION—FROM AN ART FORM TO AN ENGINEERING DISCIPLINE**

**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. Over the years, it has emerged from a pure art to a craft, and finally to an engineering discipline.

The early programmers used an adhoc programming style. This style of program development is now variously being referred to as exploratory, build and fix, and code and fixes styles.

The exploratory programming style is an informal style in the sense that there are no set rules or recommendations that a programmer has to adhere to—every programmer himself evolves his own software development techniques solely guided by his own intuition, experience, whims, and fancies. The exploratory style comes naturally to all first time programmers. Later in this chapter, we point out that except for trivial software development problems, the exploratory style usually yields poor quality and unmaintainable code and also makes program development very expensive as well as time-consuming.

As we have already pointed out, the build and fix style was widely adopted by the programmers in the early years of computing history. We can consider the exploratory program development style as an art—since this style, as is the case with any art, is mostly guided by intuition. There are many stories about programmers in the past who were like proficient artists and could write good programs using an essentially build and fix model and some esoteric knowledge. The bad programmers were left to wonder how some programmers could effortlessly write elegant and correct programs each time. In contrast, the programmers working in modern software industry rarely make use of any esoteric knowledge and develop software by applying some well-understood principles.

**Evolution Pattern for Engineering Disciplines**

If we analyse the evolution of the software development styles over the last sixty 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 pattern of evolution 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. This pattern of technology development has schematically been shown in Figure 1.1.

It can be seen from Figure 1.1 that every technology in the initial years 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, over time 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 organisation and documentation 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. As we have already pointed out, in the early days of programming, there were good programmers and bad programmers. The good programmers knew certain principles (or tricks) that helped them write good programs, which they seldom shared with the bad programmers. Program writing in later years was akin to a craft. Over the next several years, all good principles (or tricks) that were discovered by programmers along with research innovations have systematically been organised into a body of knowledge that forms the discipline of software engineering.

Software engineering principles are now being widely used in industry, 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 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 are often used successfully to develop small programs such as those written by students as classroom assignments.



**A Solution to the Software Crisis**

At present, software engineering appears to be among the few options that are available to tackle the present software crisis. But, what exactly is the present software crisis? What are its symptoms, causes, and possible solutions? To understand the present software crisis, consider the following facts. The expenses that organisations all over the world are incurring on software purchases as compared to the expenses incurred on hardware purchases have been showing an worrying trend over the years (see Figure 1.2). As can be seen in the figure, organisations are spending increasingly larger portions of their budget on software as compared to that on hardware. Among all the symptoms of the present software crisis, the trend of increasing software costs is probably the most vexing.

At present, many organisations are actually spending much more on software than on hardware. If this trend continues, we might soon have a rather amusing scenario. Not long ago, when you bought any hardware product, the essential software that ran on it came free with it. But, unless some sort of revolution happens, in not very distant future, hardware prices would become insignificant compared to software prices— when you buy any software product the hardware on which the software runs would come free with the software!!!

The symptoms of software crisis are not hard to observe. But, what are the factors that have contributed to the present software crisis? Apparently, there are many factors, the important ones being—rapidly increasing problem size, lack of adequate training in software engineering techniques, increasing skill shortage, and low productivity improvements. What is the remedy? It is believed that a satisfactory solution to the present software crisis can possibly come from a spread of software engineering practices among the developers, coupled with further advancements to the software engineering discipline itself. With this brief discussion on the evolution and impact of the discipline of software engineering, we now examine some basic concepts pertaining to the different types of software development projects that are undertaken by software companies.



**SOFTWARE DEVELOPMENT PROJECTS**

The various types of development projects that are being undertaken by software development companies, let us first understand the important ways in which professional software differs from toy software such as those written by a student in his first programming assignment.

**Programs *versus* Products**

Many toy software are being developed by individuals such as students for their classroom assignments and hobbyists for their personal use. These are usually small in size and support limited functionalities. Further, the author of a program is usually the sole user of the software and himself maintains the code. This toy software therefore usually lacks good user-interface and proper documentation.

Besides these may have poor maintainability, eﬃciency, and reliability. Since these toy software do not have any supporting documents such as users’ manual, maintenance manual, design document, test documents, etc., we call these toy software as *programs*.

In contrast, professional software usually has multiple users and, therefore, have good user-interface, proper users’ manuals, and good documentation support. Since, a software product has a large number of users; it is systematically designed, carefully implemented, and thoroughly tested. In addition, professionally written software usually consists not only of the program code but also of all associated documents such as requirements specification document, design document, test document, users’ manuals, etc. A further difference is that professional software are often too large and complex to be developed by any single individual. It is usually developed by a group of developers working in a team.

Professional software is developed by a group of software developers working together in a team. It is therefore necessary for them to use some systematic development methodology. Otherwise, they would find it very diﬃcult to interface and understand each other’s work, and produce a coherent set of documents. Even though software engineering principles are primarily intended for use in development of professional software, many results of software engineering can effectively be used for development of small programs as well. However, when developing small programs for personal use, rigid adherence to software engineering principles is often not worthwhile. An ant can be killed using a gun, but it would be ridiculously ineﬃcient and inappropriate. CAR Hoare [1994] observed that rigorously using software engineering principles to develop toy programs is very much like employing civil and architectural engineering principles to build sand castles for children to play.

**Types of Software Development Projects**

A software development company typically has a large number of on-going projects. Each of these projects may be classified into software product development projects or services type of projects. These two broad classes of software projects can be further classified into subclasses as shown in Figure 1.3

A software product development project may be either to develop a generic product or a domain specific product. A generic software product development project concerns about developing software that would be sold to a large number of customers. Since a generic software product is sold to a broad spectrum of customers, it is said to have a horizontal market. On the other hand, the services projects may either involve customizing some existing software, maintaining or developing some outsourced software. Since a specific segment of customers are targeted, these software products are said to have a vertical market. In the following, we distinguish between these two major types of software projects



**Software products**

We all know of a variety of software such as Microsoft’s Windows operating system and Oﬃce suite, and Oracle Corporation’s Oracle 8i database management software. These software are available off-the-shelf for purchase and are used by a diverse range of customers. These are called generic software products since many users essentially use the same software. These can be purchased off-the-shelf by the customers. When a software development company wishes to develop a generic product, it first determines the features or functionalities that would be useful to a large cross section of users. Based on these, the development team draws up the product specification on its own. Of course, it may base its design discretion on feedbacks collected from a large number of users. Typically, each software product is targeted to some market segment (set of users). Many companies find it advantageous to develop product lines that target slightly different market segments based on variations of essentially the same software. For example, Microsoft targets desktops and laptops through its Windows 8 operating system, while it targets high-end mobile handsets through its Windows mobile operating system, and targets servers through its Windows server operating system.

In contrast to the generic products, domain specific software products are sold to specific categories of customers and are said to have a vertical market. Domain specific software products target specific segments of customers (called *verticals*) such as banking, telecommunication, finance and accounts, and medical. Examples of domain specific software products are BANCS from TCS and FINACLE from Infosys in the banking domain and Aspen Plus from Aspen Corporation in the chemical process simulation.

**Software services**

Software services cover a large gamut of software projects such as customization, outsourcing, maintenance, testing, and consultancy. At present, there is a rapid growth in the number of software services projects that are being undertaken world-wide and software services are poised to become the dominant type of software projects. One of the reasons behind this situation is the steep growth in the available code base. Over the past few decades, a large number of programs have already been developed. Available programs can therefore be modified to quickly fulfil the specific requirements of any customer. At present, there is hardly any software project in which the program code is written from scratch, and software is being mostly developed by customizing some existing software. For example, to develop software to automate the payroll generation activities of an educational institute, the vendor may customize existing software that might has been developed earlier for a different client or educational institute.

Due to heavy reuse of code, it has now become possible to develop even large software systems

In rather short periods of time. Therefore, typical project durations are at present only a couple of months and multi-year projects have become very rare. Development of *outsourced software* is a type of software service. Outsourced software projects may arise for many reasons. Sometimes, it can make good commercial sense for a company developing a large project to outsource some parts of its development work to other companies. The reasons behind such a decision may be many. For example, a company might consider the outsourcing option, if it feels that it does not have suﬃcient expertise to develop some specific parts of the software; or it may determine that some parts can be developed cost-effectively by another company. Since an outsourced project is a small part of some larger project, outsourced projects are usually small in size and need to be completed within a few months or a few weeks of time.

The types of development projects that are being undertaken by a company can have an impact on its profitability. For example, a company that has developed a generic software product usually gets an uninterrupted stream of revenue spread over several years. However, development of a generic software product entails substantial upfront investment. Further, any return on this investment is subject to the risk of customer acceptance. On the other hand, outsourced projects are usually less risky, but fetch only one time revenue to the developing company.

**EXPLORATORY STYLE OF SOFTWARE DEVELOPMENT**

The exploratory program development style refers to an informal development style where the programmer makes use of his own intuition to develop a program rather than making use of the systematic body of knowledge categorized under the software engineering discipline. The exploratory development style gives complete freedom to the programmer to choose the activities using which to develop software.

Though the exploratory style imposes no rules a typical development starts after an initial briefing from the customer. Based on this briefing, the developers start coding to develop a working program. The software is tested and the bugs found are fixed. This cycle of testing and bug fixing continues till the software works satisfactorily for the customer. A schematic of this work sequence in a build and fix style has been shown graphically in Figure 1.4. Observe that coding starts after an initial customer briefing about what is required. After the program development is complete, a test and fix cycle continues till the program becomes acceptable to the customer. An exploratory development style can be successful when used for developing very small programs, and not for professional software. We had examined this issue with the help of the petty contractor analogy. Now let us examine this issue more carefully.



**What is wrong with the exploratory style of software development?**

Though the exploratory software development style is intuitively obvious, no software team can remain competitive if it uses this style of software development. Let us investigate the reasons behind this. In an exploratory development scenario, let us examine how do the effort and time required to develop a professional software increases with the increase in program size. Let us first consider that exploratory style is being used to develop professional software. The increase in development effort and time with problem size has been indicated in Figure 1.5. Observe the thick line plot that represents the case in which the exploratory style is used to develop a program. It can be seen that as the program size increases, the required effort and time increases almost exponentially. For large problems, it would take too long and cost too much to be practically meaningful to develop the program using the exploratory style of development. The exploratory development approach is said to break down after the size of the program to be developed increases beyond certain value. For example, using the exploratory style, you may easily solve a problem that requires writing only 1000 or 2000 lines of source code. But, if you are asked to solve a problem that would require writing one million lines of source code, you may never be able to complete it using the exploratory style; irrespective of the amount time or effort you might invest to solve it. Now observe the thin solid line plot in Figure 1.5 which represents the case when development is carried out using software engineering principles.

In this case, it becomes possible to solve a problem with effort and time that is almost linear in program size. On the other hand, if programs could be written automatically by machines, then the increase in effort and time with size would be even closer to a linear (dotted line plot) increase with size. Now let us try to understand why does the effort required to develop a program grow exponentially with program size when the exploratory style is used and then this approach to develop a program completely breaks down when the program size becomes large? To get an insight into the answer to this question, we need to have some knowledge of the human cognitive limitations (see the discussion on human psychology in subsection 1.3.1). As we shall see, the perceived (or psychological) complexity of a problem grows exponentially with its size. Please note that the perceived complexity of a problem is not related to the time or space complexity issues with which you are likely to be familiar with from a basic course on algorithms.

Even if the exploratory style causes the perceived diﬃculty of a problem to grow exponentially due to human cognitive limitations, how do the software engineering principles help to contain this exponential rise in complexity with problem size and hold it down to almost a linear increase? We will discuss in subsection 1.3.2 that software engineering principle help achieve this by profusely making use of the abstraction and decomposition techniques to overcome the human cognitive limitations. You may still wonder that when software engineering principles are used, why does the curve not become completely linear? The answer is that it is very diﬃcult to apply the decomposition and abstraction principles to completely overcome the problem complexity

.



 **Summary of the shortcomings of the exploratory style of software development:**

We briefly summarise the important shortcomings of using the exploratory development style to develop professional software:

* The foremost diﬃculty is the exponential growth of development time and effort with problem size and large-sized software becomes almost impossible using this style of development.
* The exploratory style usually results in unmaintainable code. The reason for this is that any code developed without proper design would result in highly unstructured and poor quality code.
* It becomes very diﬃcult to use the exploratory style in a team development environment. In the exploratory style, the development work is undertaken without any proper design and documentation. Therefore it becomes very diﬃcult to meaningfully partition the work among a set of developers who can work concurrently. On the other hand, team development is indispensable for developing modern software—most software mandate huge development efforts, necessitating team effort for developing these. Besides poor quality code, lack of proper documentation makes any later maintenance of the code very diﬃcult.

**Principles Deployed by Software Engineering to Overcome Human Cognitive Limitations**

Most of software engineering principles is the use of techniques to effectively tackle the problems that arise due to human cognitive limitations.

In the following subsections, with the help of Figure 1.7(a) and (b), we explain the essence of these two important principles and how they help to overcome the human cognitive limitations. In the rest of this book, we shall time and again encounter the use of these two fundamental principles in various forms and flavours in the different software development activities. A thorough understanding of these two principles is therefore needed.

**Abstraction**

Abstraction refers to construction of a simpler version of a problem by ignoring the details. The principle of constructing an abstraction is popularly known as *modelling* (or *model construction*).

When using the principle of abstraction to understand a complex problem, we focus our attention on only one or two specific aspects of the problem and ignore the rest.

Whenever we omit some details of a problem to construct an abstraction, we construct a *model* of the problem. In everyday life, we use the principle of abstraction frequently to understand a problem or to assess a situation. Consider the following two examples.

Suppose you are asked to develop an overall understanding of some country. No one in his right mind would start this task by meeting all the citizens of the country, visiting every house, and examining every tree of the country, etc. You would probably take the help of several types of abstractions to do this. You would possibly start by referring to and understanding various types of maps for that country. A map, in fact, is an abstract representation of a country. It ignores detailed information such as the specific persons who inhabit it, houses, schools, play grounds, trees, etc. Again, there are two important types of maps—physical and political maps. A physical map shows the physical features of an area; such as mountains, lakes, rivers, coastlines, and so on. On the other hand, the political map shows states, capitals, and national boundaries, etc. The physical map is an abstract model of the country and ignores the state and district boundaries.

The political map, on the other hand, is another abstraction of the country that ignores the physical characteristics such as elevation of lands, vegetation, etc. It can be seen that, for the same object (e.g. country), several abstractions are possible. In each abstraction, some aspects of the object is ignored. We understand a problem by abstracting out different aspects of a problem (constructing different types of models) and understanding them. It is not very diﬃcult to realise that proper use of the principle of abstraction can be a very effective help to master even intimidating problems.



Consider the following situation. Suppose you are asked to develop an understanding of all the living beings inhabiting the earth. If you use the naive approach, you would start taking up one living being after another who inhabit the earth and start understanding them. Even after putting in tremendous effort, you would make little progress and left confused since there are billions of living things on earth and the information would be just too much for anyone to handle. Instead, what can be done is to build and understand an abstraction hierarchy of all living beings as shown in Figure 1.8. At the top level, we understand that there are essentially three fundamentally different types of living beings—plants, animals, and fungi.

Slowly more details are added about each type at each successive level, until we reach the level of the different species at the leaf level of the abstraction tree.

A single level of abstraction can be suﬃcient for rather simple problems. However, more complex problems would need to be modelled as a hierarchy of abstractions. A schematic representation of an abstraction hierarchy has been shown in Figure 1.7(a). The most abstract representation would have only a few items and would be the easiest to understand. After one understands the simplest representation, one would try to understand the next level of abstraction where at most five or seven new information are added and so on until the lowest level is understood. By the time, one reaches the lowest level; he would have mastered the entire problem.



**Decomposition**

Decomposition is another important principle that is available in the repertoire of a software engineer to handle problem complexity. This principle is profusely made use by several software engineering techniques to contain the exponential growth of the perceived problem complexity.

The decomposition principles are popularly known as the divide and conquer principle.

A popular way to demonstrate the decomposition principle is by trying to break a large bunch of sticks tied together and then breaking them individually. Figure 1.7(b) shows the decomposition of a large problem into many small parts. However, it is very important to understand that any arbitrary decomposition of a problem into small parts would not help. The different parts after decomposition should be more or less independent of each other. That is, to solve one part you should not have to refer and understand other parts. If to solve one part you would have to understand other parts, then this would boil down to understanding all the parts together. This would effectively reduce the problem to the original problem before decomposition (the case when all the sticks tied together). Therefore, it is not suﬃcient to just decompose the problem in any way, but the decomposition should be such that the different decomposed parts must be more or less independent of each other.

As an example of a use of the principle of decomposition, consider the following.

You would understand a book better when the contents are decomposed (organised) into more or less independent chapters. That is, each chapter focuses on a separate topic, rather than when the book mixes up all topics together throughout all the pages. Similarly, each chapter should be decomposed into sections such that each section discusses a different issue. Each section should be decomposed into subsections and so on. If various subsections are nearly independent of each other, the subsections can be understood one by one rather than keeping on cross referencing to various subsections across the book to understand one.

**Why study software engineering?**

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

* The skill to participate in development of large software. You can meaningfully participate in a team effort to develop a large software only after learning the systematic techniques that are being used in the industry
* 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 various stages in software development such as 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.

**EMERGENCE OF SOFTWARE ENGINEERING**

The evolution is the result of a series of innovations and accumulation of experience about writing good quality programs. Since these innovations and programming experiences are too numerous, let us briefly examine only a few of these innovations and programming experiences which have contributed to the development of the software engineering discipline.

1. **Early Computer Programming**

Early commercial computers were very slow and too elementary as compared to today’s standards. Even simple processing tasks took considerable computation time on those computers. No wonder that programs at that time were very small in size and lacked sophistication. Those programs were usually written in assembly languages. Program lengths were typically limited to about a few hundreds of lines of monolithic assembly code. Every programmer developed his own individualistic style of writing programs according to his intuition and used this style ad hoc while writing different programs. In simple words, programmers wrote programs without formulating any proper solution strategy, plan, or design a jump to the terminal and start coding immediately on hearing out the problem. They then went on fixing any problems that they observed until they had a program that worked reasonably well. We have already designated this style of programming as the build and fix (or the exploratory programming) style.

1. **High-level Language Programming**

Computers became faster with the introduction of the semiconductor technology in the early 1960s. Faster semiconductor transistors replaced the prevalent vacuum tube-based circuits in a computer. With the availability of more powerful computers, it became possible to solve larger and more complex problems. At this time, high-level languages such as FORTRAN, ALGOL, and COBOL were introduced. This considerably reduced the effort required to develop software and helped programmers to write larger programs (why?). Writing each high-level programming construct in effect enables the programmer to write several machine instructions. Also, the machine details (registers, flags, etc.) are abstracted from the programmer. However, programmers were still using the exploratory style of software development. Typical programs were limited to sizes of around a few thousands of lines of source code.

1. **Control Flow-based Design**

As the size and complexity of programs kept on increasing, the exploratory programming style proved to be insuﬃcient. Programmers found it increasingly diﬃcult not only to write cost-effective and correct programs, but also to understand and maintain programs written by others. To cope up with this problem, experienced programmers advised other programmers to pay particular attention to the design of a program’s control flow structure.

In order to help develop programs having good control flow structures, the flow charting technique was developed. Even today, the flow charting technique is being used to represent and design algorithms; though the popularity of the flow charting technique to represent and design programs has diminished due to the emergence of more advanced techniques more advanced techniques.

Figure 1.9 illustrates two alternate ways of writing program code for the same problem. The flow chart representations for the two program segments of Figure 1.9 are drawn in Figure 1.10. Observe that the control flow structure of the program segment in Figure 1.10(b) is much simpler than that of Figure 1.10(a). By examining the code, it can be seen that Figure 1.10(a) is much harder to understand as compared to Figure 1.10(b).



This example corroborates the fact that if the flow chart representation is simple, then the corresponding code should be simple. You can draw the flow chart representations of several other problems to convince yourself that a program with complex flow chart representation is indeed more diﬃcult to understand and maintain.



Let us now try to understand why a program having good control flow structure would be easier to develop and understand. In other words, let us understand why a program with a complex flow chart representation is diﬃcult to understand? The main reason behind this situation is that normally one understands a program by mentally tracing its execution sequence (i.e. statement sequences) to understand how the output is produced from the input values. That is, we can start from a statement producing an output, and trace back the statements in the program and understand how they produce the output by transforming the input data. Alternatively, we may start with the input data and check by running through the program how each statement processes (transforms) the input data until the output is produced. For example, for the program of Figure 1.10(a) you would have to understand the execution of the program along the paths 1-2-3-7-8-10, 1-4-5-6-9-10, and1-4-5-2-3-7-8-10.

A program having a messy control flow (i.e. flow chart) structure would have a large number of execution paths (see Figure 1.11). Consequently, it would become extremely diﬃcult to determine all the execution paths, and tracing the execution sequence along all the paths trying to understand them can be nightmarish. It is therefore evident that a program having a messy flow chart representation would indeed be diﬃcult to understand and debug.

**Are GO TO statements the culprits?**

In a landmark paper, Dijkstra [1968] published his (now famous) article “GO TO Statements Considered Harmful”. He pointed out that unbridled use of GO TO statements is the main culprit in making the control structure of a program messy. To understand his argument, examine Figure

1.11 Which shows the flow chart representation of a program in which the programmer has used rather too many GO TO statements. GO TO statements alter the flow of control arbitrarily, resulting in too many paths. But, then why does use of too many GOTO statements make a program hard to understand? Soon it became widely accepted that good programs should have very simple control structures. It is possible to distinguish good programs from bad programs by just visually examining their flow chart representations. The use of flow charts to design good control flow structures of programs became wide spread.



**Structured programming—a logical extension**

The need to restrict the use of GO TO statements was recognised by everybody. However, many programmers were still using assembly languages. JUMP instructions are frequently used for program branching in assembly languages. Therefore, programmers with assembly language programming background considered the use of GO TO statements in programs inevitable. However, it was conclusively proved by Bohm and Jacopini [1966] that only three programming constructs—sequence, selection, and iteration—were suﬃcient to express any programming logic. This was an important result—it is considered important even today.

An example of a sequence statement is an assignment statement of the form a=b;. Examples of selection and iteration statements are the if-then-else and the do-while statements respectively. Gradually, everyone accepted that it is indeed possible to solve any programming problem without using GO TO statements and that indiscriminate use of GO TO statements should be avoided. This formed the basis of the structured programming methodology.

Structured programs avoid unstructured control flows by restricting the use of GO TO statements. Structured programming is facilitated, if the programming language being used supports single-entry, single-exit program constructs such as if-then-else, do-while, etc. Thus, an important feature of structured programs is the design of good control structures. An example illustrating this key difference between structured and unstructured programs is shown in Figure 1.9. The program in Figure 1.9(a) makes use of too many GO TO statements, whereas the program in Figure 1.9(b) makes use of none.

The control flow diagram of the program making use of GO TO statements is obviously much more complex as can be seen in Figure 1.10.Besides the control structure aspects, the term *structured program* is being used to denote a couple of other program features as well. A structured program should be modular. A modular program is one which is decomposed into a set of modules1 such that the modules should have low interdependency among each other.

But, what are the main advantages of writing structured programs compared to the unstructured ones? Research experiences have shown that programmers commit less number of errors while using structured if-then-else and do-while statements than when using test-and-branch code constructs. Besides being less error-prone, structured programs are normally more readable, easier to maintain, and require less effort to develop compared to unstructured programs. The virtues of structured programming became widely accepted and the structured programming concepts are being used even today. However, violations to the structured programming feature are usually permitted in certain specific programming situations, such as exception handling, etc.

Very soon several languages such as PASCAL, MODULA, C, etc., became available which were specifically designed to support structured programming. These programming languages facilitated writing modular programs and programs having good control structures. Therefore, messy control structure was no longer a big problem. So, the focus shifted from designing good control structures to designing good data structures for programs.

1. **Data Structure-oriented Design**

Computers became even more powerful with the advent of integrated circuits (ICs) in the early seventies. These could now be used to solve more complex problems. Software developers were tasked to develop larger and more complicated software. This often required writing in excess of several tens of thousands of lines of source code. The control flow-based program development techniques could not be used satisfactorily any more to write those programs, and more effective program development techniques were needed.

It was soon discovered that while developing a program, it is much more important to pay attention to the design of the important data structures of the program than to the design of its control structure. Design techniques based on this principle are called data structure-oriented

 Design techniques.

In the next step, the program design is derived from the data structure. An example of a data structure-oriented design technique is the Jackson’s Structured Programming (JSP) technique developed by Michael Jackson [1975]. In JSP methodology, a program’s data structure is first designed using the notations for sequence, selection, and iteration. The JSP methodology provides an interesting technique to derive the program structure from its data structure representation.

Several other data structure-based design techniques were also developed. Some of these techniques became very popular and were extensively used. Another technique that needs special mention is the Warnier-Orr Methodology [1977, 1981]. However, we will not discuss these techniques in this text because now-a-days these techniques are rarely used in the industry and have been replaced by the data flow-based and the object-oriented techniques.

1. **Data Flow-oriented Design**

As computers became still faster and more powerful with the introduction of very large scale integrated (VLSI) Circuits and some new architectural concepts, more complex and sophisticated software were needed to solve further challenging problems. Therefore, software developers looked out for more effective techniques for designing software and soon data flow-oriented techniques were proposed.

The functions (also called as processes) and the data items that are exchanged between the different functions are represented in a diagram known as a data flow diagram (DFD).The program structure can be designed from the DFD representation of the problem.

**DFDs: A crucial program representation for procedural program design**

DFD has proven to be a generic technique which is being used to model all types of systems, and not just software systems. For example, Figure 1.12 shows the data-flow representation of an automated car assembly plant. If you have never visited an automated car assembly plant, a brief description of an automated car assembly plant would be necessary. In an automated car assembly plant, there are several processing stations (also called workstations) which are located along side of a conveyor belt (also called an assembly line). Each workstation is specialised to do jobs such as fitting of wheels, fitting the engine, spray painting the car, etc. As the partially assembled program moves along the assembly line, different workstations perform their respective jobs on the partially assembled software. Each circle in the DFD model of Figure 1.12 represents a workstation (called a process or bubble). Each workstation consumes certain input items and produces certain output items. As a car under assembly arrives at a workstation, it fetches the necessary items to be fitted from the corresponding stores (represented by two parallel horizontal lines), and as soon as the fitting work is complete passes on to the next workstation. It is easy to understand the DFD model of the car assembly plant shown in Figure 1.12 even without knowing anything regarding DFDs. In this regard, we can say that a major advantage of the DFDs is their simplicity. In Chapter 6, we shall study how to construct the DFD model of a software system. Once you develop the DFD model of a problem, data flow-oriented design techniques provide a rather straight forward methodology to transform the DFD representation of a problem into an appropriate software design.



1. **Object-oriented Design**

Data flow-oriented techniques evolved into object-oriented design (OOD) techniques in the late seventies. Object-oriented design technique is an intuitively appealing approach, where the natural objects (such as employees, pay-roll-register, etc.) relevant to a problem are first identified and then the relationships among the objects such as composition, reference, and inheritance are determined. Each object essentially acts as a data hiding (also known as data abstraction) entity. Object-oriented techniques have gained wide spread acceptance because of their simplicity, the scope for code and design reuse, promise of lower development time, lower development cost, more robust code, and easier maintenance.

**NOTABLE CHANGES IN SOFTWARE DEVELOPMENT PRACTICES**

The exploratory style of software development and another development effort based on modern software engineering practices. The following noteworthy differences between these two software development approaches would be immediately observable.

* An important difference is that the exploratory software development style is based on error correction (build and fix) while the software engineering techniques are based on the principles of error prevention. Inherent in the software engineering principles is the realisation that it is much more cost-effective to prevent errors from occurring than to correct them as and when they are detected. Even when mistakes are committed during development, software engineering principles emphasize detection of errors as close to the point where the errors are committed as possible). In the exploratory style, errors are detected only during the final product testing. In contrast, the modern practice of software development is to develop the software through several well defined stages such as requirements specification, design, coding, testing, etc., and attempts are made to detect and fix as many errors as possible in the same phase in which they are made.
* In the exploratory style, coding was considered synonymous with software development. For instance, this naive way of developing a software believed in developing a working system as quickly as possible and then successively modifying it until it performed satisfactorily. Exploratory programmers literally dive at the computer to get started with their programs even before they fully learn about the problem!!! It was recognised that exploratory programming not only turns out to be prohibitively costly for non-trivial problems, but also produces hard-to-maintain programs. Even minor modifications to such programs later can become nightmarish. In the modern software development style, coding is regarded as only a small part of the overall software development activities. There are several development activities such as design and testing which may demand much more effort than coding.
* A lot of attention is now being paid to requirements specification. Significant effort is being devoted to develop a clear and correct specification of the problem before any development activity starts. Unless the requirements specification is able to correctly capture the exact customer requirements, large number of rework would be necessary at a later stage. Such rework would result in higher cost of development and customer dissatisfaction.
* Now there is a distinct design phase where standard design techniques are employed to yield coherent and complete design models.

* Periodic reviews are being carried out during all stages of the development process. The main objective of carrying out reviews is phase containment of errors, i.e. detect and correct errors as soon as possible. Phase containment of errors is an important software engineering principle. We will discuss this technique in Chapter 2.
* Today, software testing has become very systematic and standard testing techniques are available. Testing activity has also become all encompassing, as test cases are being developed right from the requirements specification stage.
* There is better visibility of the software through various developmental activities. In the past, very little attention was being paid to producing good quality and consistent documents. In the exploratory style, the design and test activities, even if carried out (in whatever way), were not documented satisfactorily. Today, consciously good quality documents are being developed during software development. This has made fault diagnosis and maintenance far smoother. We will see in Chapter 3 that in addition to facilitating product maintenance, increased visibility makes management of a software project easier.
* Now, projects are being thoroughly planned. The primary objective of project planning is to ensure that the various development activities take place at the correct time and no activity is halted due to the want of some resource. Project planning normally includes preparation of various types of estimates, resource scheduling, and development of project tracking plans. Several techniques and automation tools for tasks such as configuration management, cost estimation, scheduling, etc., are being used for effective software project management.
* Several metrics (quantitative measurements) of the products and the product development activities are being collected to help in software project management and software quality assurance.