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.

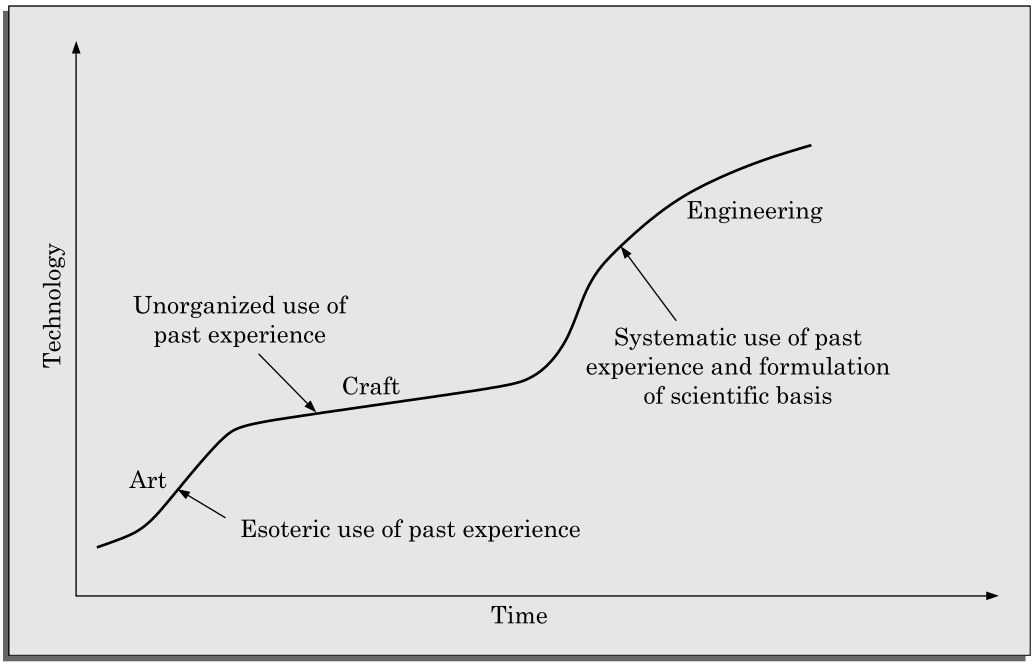

Figure 1.1: Evolution of technology with time.

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.

1.1.2 A Solution to the Software Crisis

As we have already pointed out, 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 solution? To understand the present software crisis, consider the following facts:

The expenses that organizations all around the world are incurring on software purchases as compared to those on hardware purchases have been showing a worrying trend over the years (see Figure 1.2). Organizations are spending larger and larger portions of their budget on software as compared to hardware. Not only are the software products turning out to be more expensive than hardware, but they also present a host of other problems to the customers: software products are difficult to alter, debug, and enhance; use resources non-optimally; often fail to meet the user requirements; are far from being reliable; frequently crash; and are often delivered late. Among all these, the trend of increasing software costs is probably the most important symptom of the present software crisis.

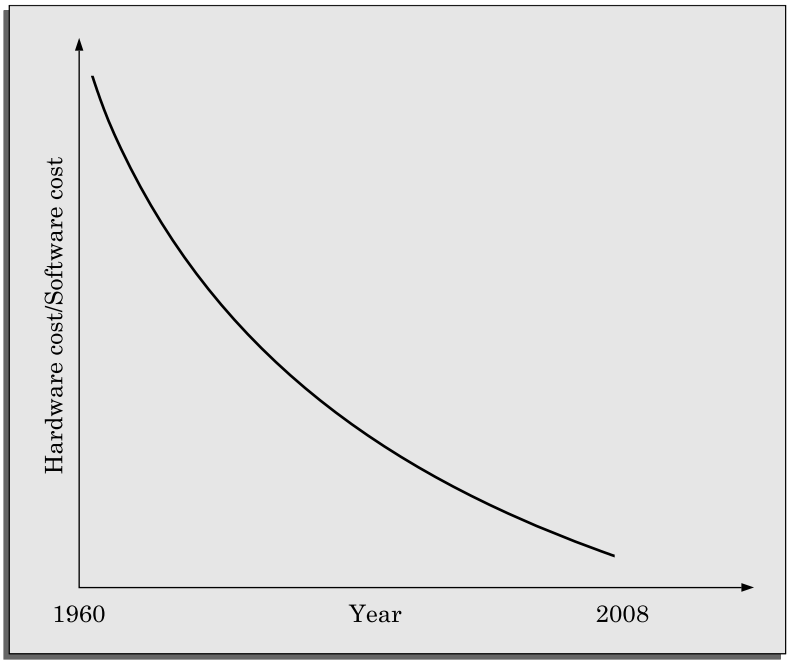

Figure 1.2: Relative changes of hardware and software costs over time.

At present, many customers are actually spending much more on buying software than on buying 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 a 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!!! But, which factors have precipitated the present software crisis? Apparently, there are many factors, the important ones being: rapidly increasing problem sizes, 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 engineers, 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 software development projects.

1.2 SOFTWARE DEVELOPMENT PROJECTS

Before discussing the software engineering principles, it is important to form some ideas on the software projects that companies undertake to develop commercially useful software. We all use commercial software such as Microsoft Windows, Oracle DBMS, software accompanying a camcorder or a laser printer, etc. There are also more sophisticated commercial software such as railways reservation system, nuclear power plan control software, etc. with which we might not be familiar. All these software are developed as software projects in industry. First, let us understand how these commercial software different from the programs that students develop as their programming assignments.

1.2.1 Programs versus Products

Programs are developed by individuals for their personal use. They are, therefore, small in size and have limited functionality. Further, the author of a program is usually the sole user of the program and himself maintains the program. These, therefore, usually lack good user-interface and proper documentation. For example, the programs developed by a student as part of his class assignments are programs and not software products. On the other hand, software products have 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, a software product 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 the software products are often too large to be developed by any single individual. It is usually developed by a group of engineers working in a team.

We shall distinguish between *software engineers* who develop software products and *programmers* who write programs. Since a group of software engineers usually work together in a team to develop a software product, it is necessary for them to adopt 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 software products, 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] has rightly 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.

1.2.2 Types of Software Development Projects

A software development company is typically structured into a large number of development teams. Each team either develops some software product or handles some outsourced projects. Let us now discuss a little more about these two types of software development work that different teams undertake. First let us discuss about software products.

Software product development projects

A software product can either be a generic or a custom developed product. Generic products are used by a large and diverse range of customers. Examples of generic products are operating systems such as Microsoft Windows, database management systems such as Oracle, and banking software products such as Finacle; just to name a few. A company developing a generic product, first determines what may be useful to a large cross-section of users. It then draws up the product specification on its own, possibly based on feedbacks collected from a large number of users. On the other hand, a customized software product is developed according to the specification drawn up by one or at most a few customers. Usually, companies develop customized software by tailoring some of their existing products. For example, when an academic institution wishes to have a software that would automate its important activities such as student registration, grading, fee collection, etc. Companies would normally develop such a software as a customized product. This means that the company developing this software would normally tailor one of its existing software products that it might have developed in the past for some other academic institution.

Outsourced projects

Sometimes, it can make good commercial sense for a company developing a software product to outsource some part of its work as a project to another company. There might be various reasons for such outsourcing. For example, a company might consider the outsourcing option, if it feels that it does not have sufficient expertize to develop some specific parts of the product; or it may determine that some parts can be developed more cost-effectively by another company. Since an outsourced project is a small part of some product development, all outsourced projects are usually small in size and are normally completed within a year or so.

1.2.3 Software Projects being Undertaken by Indian Companies

Indian software industries have excelled in executing outsourced software projects and have made a name for themselves all over the world. Of late, the Indian companies have slowly started to focus on product development as well (which was the front they were lacking in till date).

The types of development work being handled 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 over several years. On the other hand, outsourced projects fetch a one time revenue to the developing company.

1.3 WHAT IS WRONG WITH THE EXPLORATORY STYLE OF SOFTWARE DEVELOPMENT?

We have already seen that the exploratory program development style comes naturally to everybody. But, its successful application is limited to the development of very small programs only. We had examined this issue with the help of the petty contractor analogy. Now let us examine this issue more carefully. How exactly do the effort and time required to develop a product increase with the increase in product size? First consider the case where exploratory style is used to develop a product. The increase in development effort and time with problem size has been indicated in Figure 1.3. Observe the thick line plot representing the case when the exploratory style is used to develop a product. As the product size increases, the required effort and time increase almost exponentially. For large problems, it becomes almost impossible to develop the product 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 requiring only 1000 or 2000 lines of source code to be written. But, if you are asked to solve a problem that requires 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 invest to solve it. Now observe Figure 1.3 for the case when development is carried out using software engineering principles (thin solid line plot). In this case, it becomes possible to solve a problem with effort that is almost linear in product 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 rise (dotted line plot).

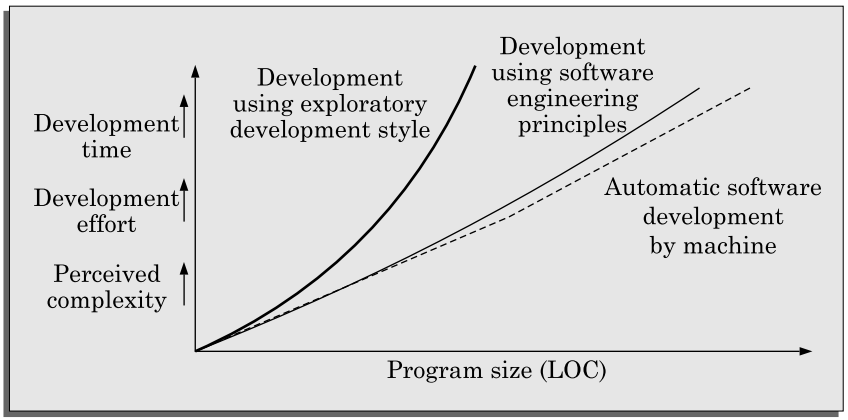

Figure 1.3: Increase in development time and effort with problem size.

But, why does the effort required to develop a product grow exponentially with product size when the exploratory style is used and then the approach completely breaks down when the product 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. The perceived complexity of a problem that

we are talking about here is not related to the time or space complexity issues with which you are most likely to be familiar with.

> The psychological or perceived complexity of a problem concerns the difficulty level experienced by a programmer while solving it using the exploratory development style.

Even if the exploratory style cause 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 a linear increase? We shall learn in subsection 1.3.2 that software engineering principles 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: it is very difficult to apply the decomposition and abstraction principles perfectly.

Besides the exponential growth of development time and effort with problem size, the exploratory development approach suffers from several other difficulties as well. The exploratory style usually results in unmaintainable code. The reason for this is that any code developed without proper design and with a build and fix methodology would result in highly unstructured and poor quality code. Further, it becomes very difficult to use the exploratory style in a team development environment. In the exploratory style, the development is carried out without any proper design and documentation. Therefore, it becomes very difficult to meaningfully partition the work among a set of developers. On the other hand, team development is indispensable for developing modern software products — most software products mandate huge development efforts, necessitating team effort for developing these.

1.3.1 Perceived Problem Complexity: An Interpretation Based on Human Cognition Mechanism

The rapid increase of the perceived complexity of a problem with problem size can be explained from an interpretation of the human cognition mechanism. A simple understanding of the human cognition mechanism would also give us an insight into why the exploratory style of development leads to undue increase in the time and effort required to develop a programming solution. It can also explain why it quickly becomes practically infeasible to solve problems larger than a certain size; whereas, using software engineering principles, the required effort grows almost linearly with size (as indicated by the thin solid line in Figure 1.3).

Psychologists say that the human memory can be thought to be made up of two distinct parts [Miller, 56]: the short-term and long-term memories. A schematic representation of these two types of memories and their roles in human cognition mechanism has been shown in Figure 1.4. We now elaborate this human cognition model as follows:

Short-term memory

The short-term memory, as the name itself suggests, can store information for a short while— usually up to a few seconds, and at most for a few minutes. The short-term memory is also sometimes referred to as the working memory. The information stored in the short-term memory is immediately accessible for processing by the brain. The short-term memory of an

average person can store up to seven items; but in extreme cases it can vary anywhere from five to nine items (7 ± 2). As shown in Figure 1.4, the short-term memory participates in all interactions of the human mind with its environment.

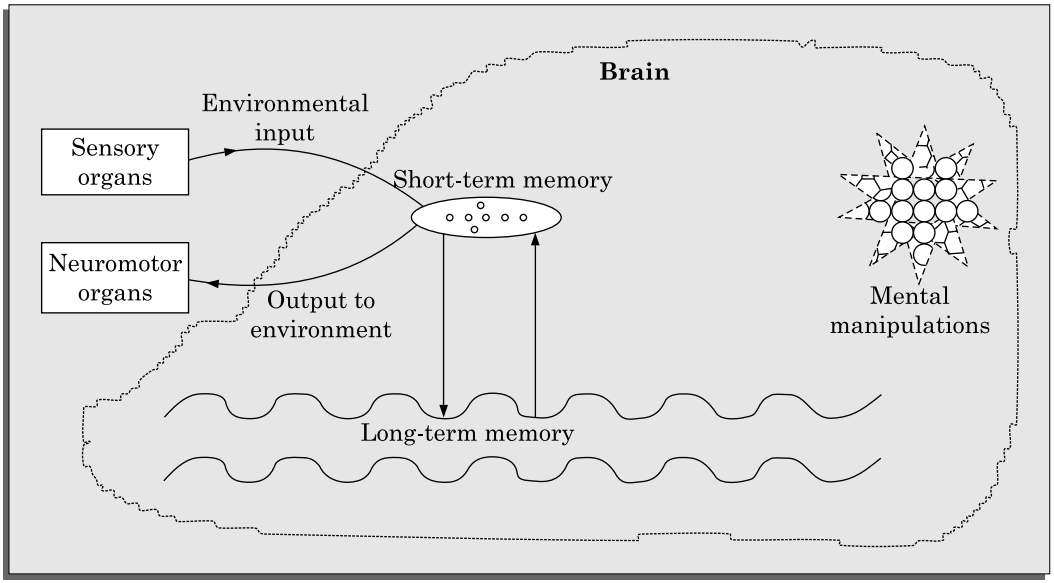

Figure 1.4: A simple model of human cognition mechanism.

It should be clear that the short-term memory plays a very crucial part in the human cognition mechanism. All information collected through the sensory organs (eye, ear, touch, smell, taste) are first stored in the short-term memory. The short-term memory is also used by the brain to drive the neuromotor organs (hand, finger, feet, etc.). The mental manipulation unit also gets its inputs from the short-term memory and stores back any output it produces. Further, information retrieved from the long-term memory first gets stored in the short-term memory. For example, if you are asked the question: "If it is 10 a.m. now, how many hours are remaining today?" First, 10 a.m. would be stored in the short-term memory. Next, the information that a day is 24 hours long would be fetched from the long-term memory into the short-term memory. The mental manipulation unit would compute the difference (24-10), and 14 hours would get stored in the short-term memory. As you can notice, this model is very similar to the organization of a computer in terms of cache, main memory and processor.

An item stored in the short-term memory can get lost either due to decay with time or displacement by newer information. This restricts the duration for which an item is stored in the short-term memory to a few tens of seconds. However, an item can be retained longer in the short-term memory by recycling. That is, when we repeat or refresh an item consciously, we can remember it for a much longer duration. Certain information stored in the short-term memory, under certain circumstances gets stored in the long-term memory.

Long-term memory

Unlike the short-term memory, the size of the long-term memory is not known to have a definite upper bound. The size of the long-term memory can vary from several million items

to several billion items, largely depending on how actively a person exercises his mental faculty. An item once stored in the long-term memory, is usually retained for several years. But, how do items get stored in the long-term memory? Items present in the short-term memory can get stored in the long-term memory either through large number of refreshments (repetitions) or by forming links with already existing items in the long-term memory. For example, you possibly remember your own telephone number because you might have repeated (refreshed) it for a large number of times in your short-term memory. Let us now take an example of a situation where you may form links to existing items in the long-term memory to remember certain information. Suppose you want to remember the 10 digit mobile number 9433795369. To remember it by rote may be intimidating. But, suppose you consider the number as split into 9433 7953 69 and notice that 94 is the code for BSNL, 33 is the code for Kolkata, suppose 79 is your year of birth, and 53 is your roll number, and the rest of the four numbers are each one less than the corresponding digits of the previous number; you have effectively established links with already stored items, making it easier to remember the number.

Item

We have so far only mentioned the number of items that the long-term and the short-term memories can store. But, what exactly is an item? An item is any set of related information. According to this definition, a character such as 'a' or a digit such as '5' can be items. A word, a sentence, a story or even a picture can each be a single item. Each item normally occupies one place in memory. The definition of an item as any set of related information implies that when you are able to relate several different items together, the information that should normally occupy several places can be stored using only one place in the memory. This phenomenon of forming one item from several items is referred to as chunking by psychologists. For example, if you are given the binary number 110010101001 — it may prove very hard for you to understand and remember. But, the octal form of the number 6251 (i.e. the representation as (110)(010)(101)(001)) may be much easier to understand and remember since you have managed to create chunks of three items each.

Evidence of short-term memory

Evidences of short-term memory manifest themselves in many of our day-to-day experiences. As an evidence of the short-term memory, consider the following situation. Suppose, you look up a number from the telephone directory and start dialling it. If you find the number to be busy, you would dial the number again after a few seconds—in this case, you would be able to do so almost effortlessly without having to look up the directory. But, after several hours or days since you dialled the number last, you may not remember the number at all, and would need to consult the directory again.

The magical number 7

Miller called the number seven as the magical number [Miller, 56] since if a person deals with seven or less number of unrelated information at a time these would be easily accommodated in the short-term memory. So, he can easily understand it. As the number of new information one has to deal with increases beyond seven, it becomes exceedingly difficult to understand it. This observation can be easily extended to writing programs.

When the number of details (or variables) that one has to track to solve a problem increases beyond seven, it exceeds the capacity of the short-term memory and it becomes exceedingly more difficult for a human mind to grasp the problem.

A small program having just a few variables is within the easy grasp of an individual. As the number of independent variables in the program increases, it quickly exceeds the grasping power of an individual and would require an unduly large effort to master the problem. This outlines a possible reason behind the exponential nature of the effort-size plot (thick line) shown in Figure 1.3. The situation depicted in Figure 1.3 arises mostly due to the human cognitive limitations. Instead of a human, if a machine could be writing (generating) a program, the slope of the curve would be linear, as the cache size (short-term memory) of a computer is quite large. But, why does the effort-size curve become almost linear when software engineering principles are deployed? This is because software engineering principles extensively use the techniques that are designed specifically to overcome the human cognitive limitations. We discuss this issue in the next subsection.

1.3.2 Principles Deployed by Software Engineering to Overcome Human Cognitive Limitations

We shall see throughout this book that a central theme of most of software engineering principles is the use of techniques to effectively tackle the problems that arise due to human cognitive limitations.

Mainly two important principles are deployed by software engineering principles to overcome the problems arising due to human cognitive limitations. These two principles are **abstraction** and **decomposition**.

In the following, 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 details. The principle of abstraction is popularly known as **modelling** (or model construction).

Simplifying a problem by omitting unnecessary details is known as the **principle of abstraction**.

To use the principle of abstraction to understand a complex problem, we must focus our attention on only one aspect of the problem each time, and ignore details that are not related to the aspect we are focusing. Whenever we omit some details of a problem, we get a model of the problem. In everyday life, we use the principle of abstraction frequently to understand a problem well or to assess a situation. Consider the following example: suppose you are asked

to develop an overall understanding of some country. No one would begin 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 of that country. A map, in fact, is an abstract representation of a country. It ignores detailed information such as people who inhabit it, houses, playgrounds, 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 state, country and national boundaries. 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 realize that proper use of the principle of abstraction can be a very effective help to master even intimidating problems.

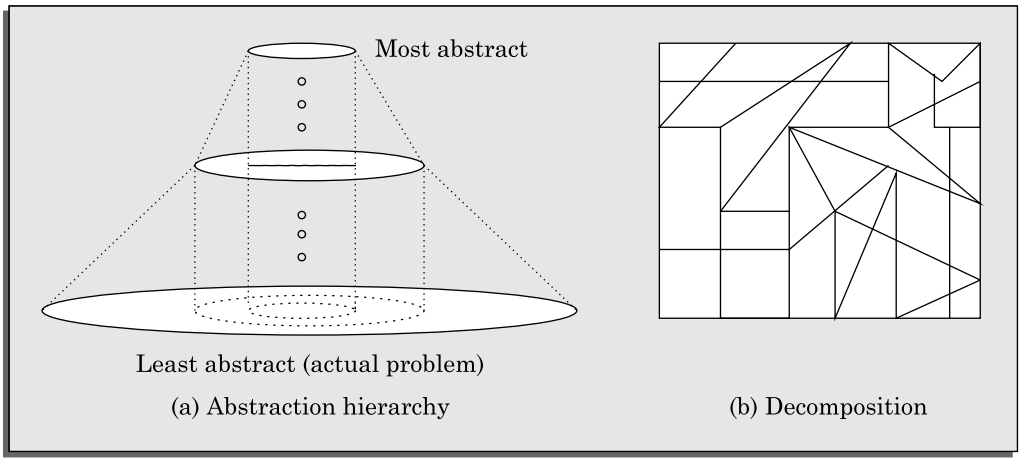

Figure 1.5: Schematic representation.

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 1.5(a). The most abstract representation would have only a few items and would be the easiest to understand. After one understands the simplest representation, he 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.

Though we might not have noticed the use, we do frequently use the principle of abstraction in many day-to-day applications. 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 any one to handle. Instead, what can be done is to build and understand an abstraction hierarchy of all living beings as shown in Figure 1.6. 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.

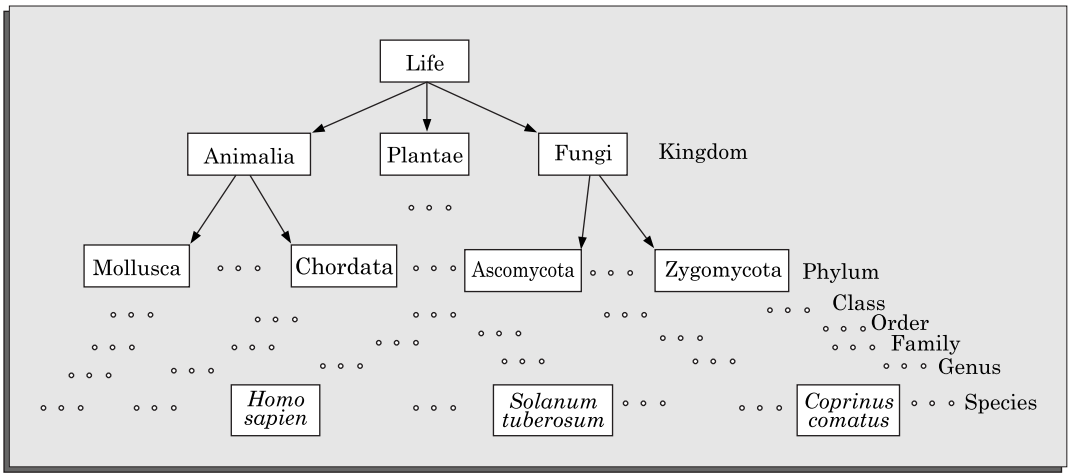

Figure 1.6: An abstraction hierarchy classifying living organisms.

Decomposition

Decomposition is another principle that is useful to handle complexity in a problem. This principle is also profusely made use of by the software engineering principles to contain the exponential growth of the perceived problem complexity. The decomposition principle is popularly known as the divide and conquer principle.

> The decomposition principle advocates decomposing the problem into many small independent parts. The small parts are then taken up one by one and solved separately. The idea is that each small part would be easy to grasp and understand and can be easily solved. The full problem is solved when all the parts are solved.

A popular way to demonstrate the decomposition principle is by trying to break a large bunch of sticks tied together versus breaking them individually. Figure 1.5(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 component 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 (organized) 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 issues. Each section should be decomposed into subsections, and so on.

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.

1.4 EMERGENCE OF SOFTWARE ENGINEERING

We have already pointed out that software engineering techniques have evolved over many years in the past. This evolution is the result of a series of innovations and accumulation of experience about writing good quality programs. Since these 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.

1.4.1 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 and 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 product that worked reasonably well. We have already designated this style of programming as the *build and fix* (or the exploratory programming) style.

1.4.2 High-Level Language Programming

Computers became faster with the introduction of the semiconductor technology in the early 60's. 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 products 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.

1.4.3 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*.

> A program's control flow structure indicates the sequence in which the program's instructions are executed.

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

Figures 1.7(a) and (b) illustrate two alternate ways of writing program code for the same problem. The flow chart representations for the two program segments of Figure 1.7 are drawn in Figure 1.8. Observe that the control flow structure of the program segment in Figure 1.8(b) is much more simpler than that of Figure 1.8(a). 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.

```
1       if(customer_savings_balance>withdrawal_request) {
2   100:    issue_money=TRUE;
3           GOTO 110;
        }
4       else if(privileged_customer==TRUE)
5           GOTO 100;
6       else GOTO 120;
7   110: activate_cash_dispenser(withdrawal_request);
8       GOTO 130;
9   120:    print(error);
10  130:    end–transaction();
```

(a) An example unstructured program

```
1   if(privileged_customer||(customer_savings_balance>withdrawal_request)){
2           activate_cash_dispenser(withdrawal_request);
    }
3   else print(error);
4   end–transaction();
```

(b) Corresponding structured program

Figure 1.7: Two programs.

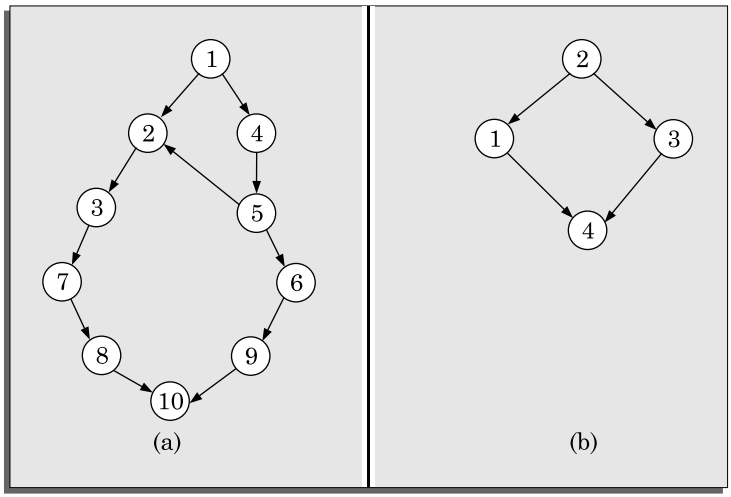

Figure 1.8: Control flow graphs of the programs in Figures 1.7(a) and (b).

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 1.8(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, and 1-4-5-2-3-7-8-10. A program having a messy control flow (i.e. flow chart) structure, would have a large number of execution paths [see Figure 1.8(a)]. 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". It was pointed out by him that unbridled use of GO TO statements is the main culprit in making the control structure of a program messy. To understand his argument, look at Figure 1.9 which shows the flow chart representation of a program in which the programmer used rather too many GO TO statements. GO TO statements alter the flow of control arbitrarily, resulting in too many paths. A programmer trying to understand the program would have to trace and understand the processing that take place along all these paths making program understanding and debugging extremely complicated. Soon it became widely accepted that good programs should have neat control structures. It 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 widespread.

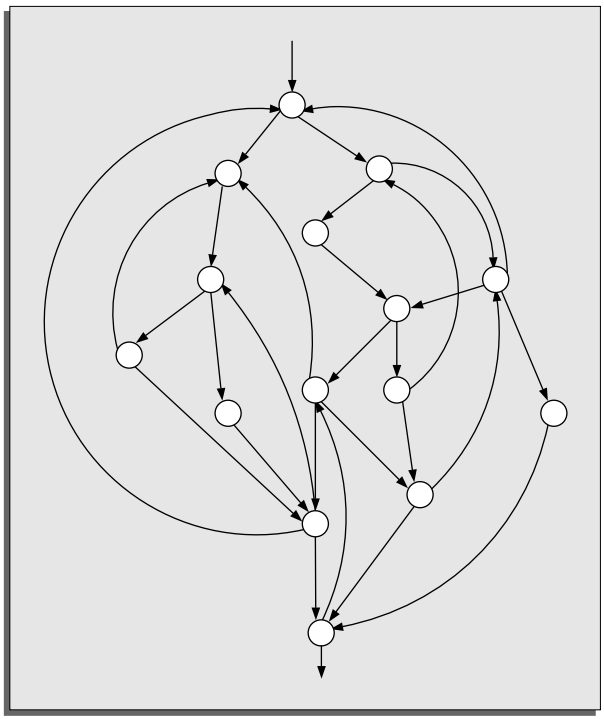

Figure 1.9: CFG of a program using too many GO TO statements.

A logical extension: structured programming

The need to restrict the use of GO TO statements was recognized 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 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.

A program is called structured when it uses only the sequence, selection, and iteration types of constructs and is modular.

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 1.7. The program in Figure 1.7(a) makes use of too many GO TO statements, whereas the program in Figure 1.7(b) makes use of none. The flow chart of the program making use of GO TO statements is obviously much more complex as can be seen in Figure 1.8.

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 modules[1] such that the modules should have low interdependency among each other. We discuss the concept of modular programs in Chapter 5.

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 are being used even today. However, violations to the structured programming feature is 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]In this text, we shall use the terms *module* and *module structure* to loosely mean the following: A module is a collection of procedures and data structures accessible only to those procedures. A module forms an independently compiled unit and may be linked to other modules to form a complete application. The term module structure will be used to denote the way in which different modules invoke each other's procedures.

1.4.4 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 engineers were tasked to develop larger and more complicated software products which 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 was 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**.

Using data structure-oriented techniques, first a program's data structures are designed.

In the next step, the program design is derived from the data structure. An example of a data structure-oriented design technique is the JSP (Jackson's Structured Programming) 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 nowadays these techniques are rarely used in the industry and have been replaced by the data flow-based and the object-oriented techniques.

1.4.5 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 products were needed to solve further challenging problems. Therefore, software engineers looked out for more effective techniques for designing software products and soon **data flow-oriented techniques** were proposed.

The data flow-oriented techniques advocate that the major data items handled by a system must first be identified and then the processing required on these data items to produce the desired outputs should be determined.

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 1.10 shows the data-flow representation of