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.

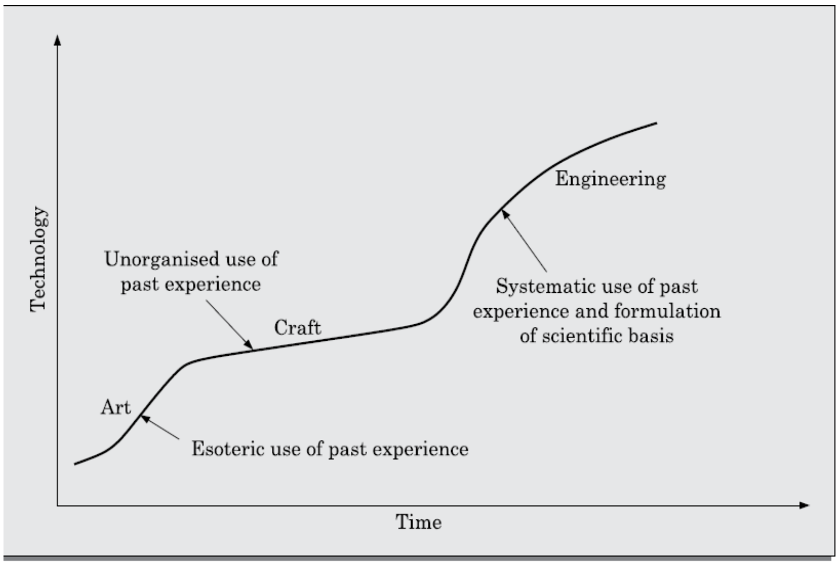

Technology
Time
Art
Esoteric use of past experience
Unorganised use of past experience
Craft
Engineering
Systematic use of past experience and formulation of scientific basis

FIGURE 1.1 Evolution of technology with time.

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.

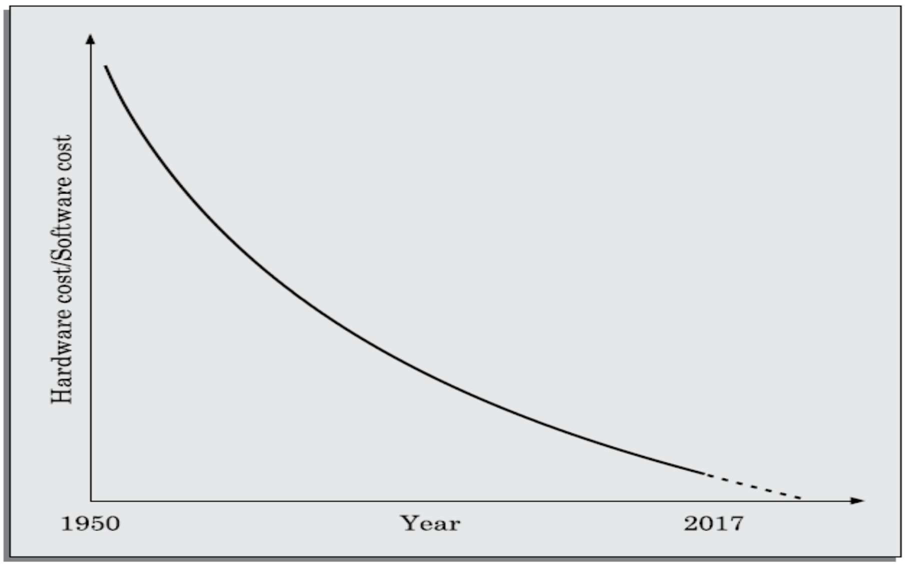


FIGURE 1.2 Relative changes of hardware and software costs over time.

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

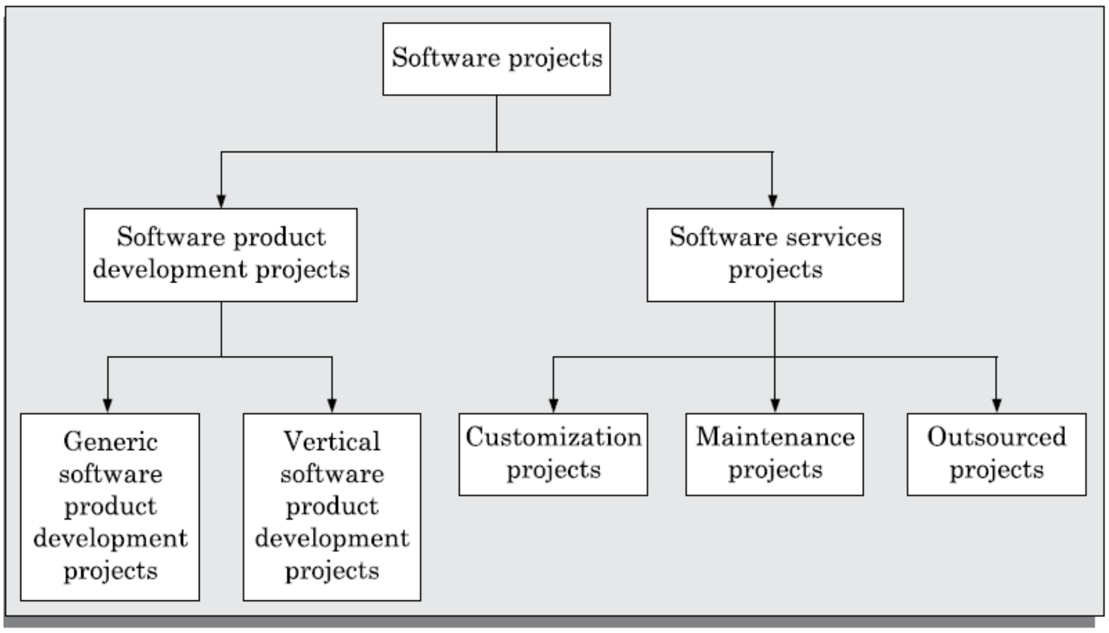

Software projects
Software product development projects
Software services projects
Generic software product development projects
Vertical software product development projects
Customization projects
Maintenance projects
Outsourced projects

FIGURE 1.3 A classification 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.

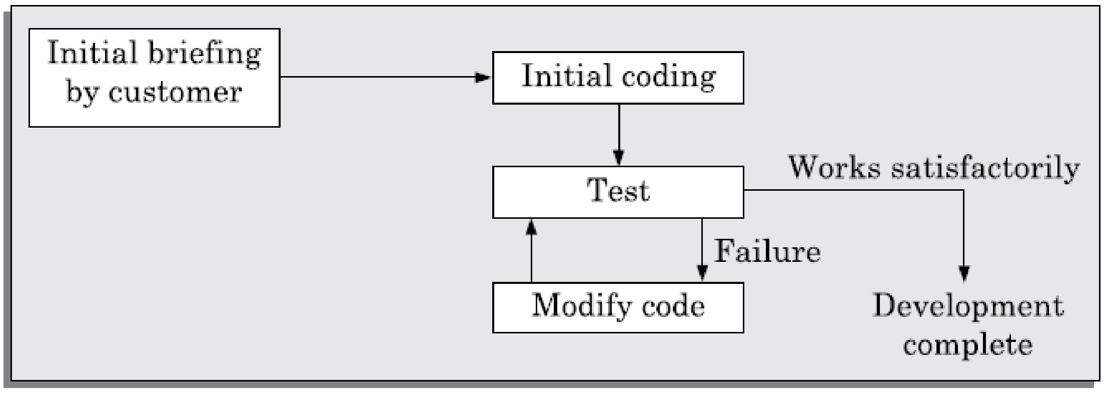


FIGURE 1.4 Exploratory program development.

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

.

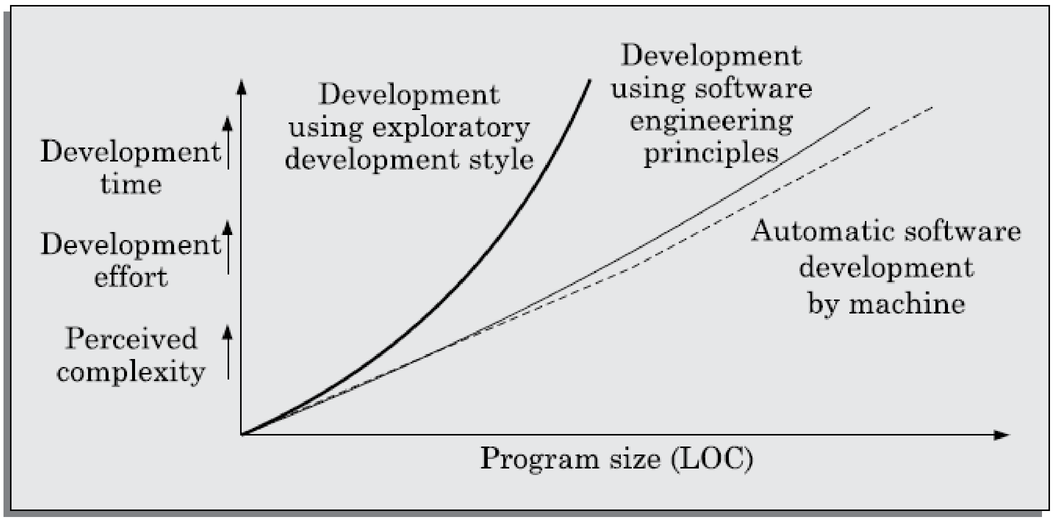


FIGURE 1.5 Increase in development time and effort with problem size.

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.

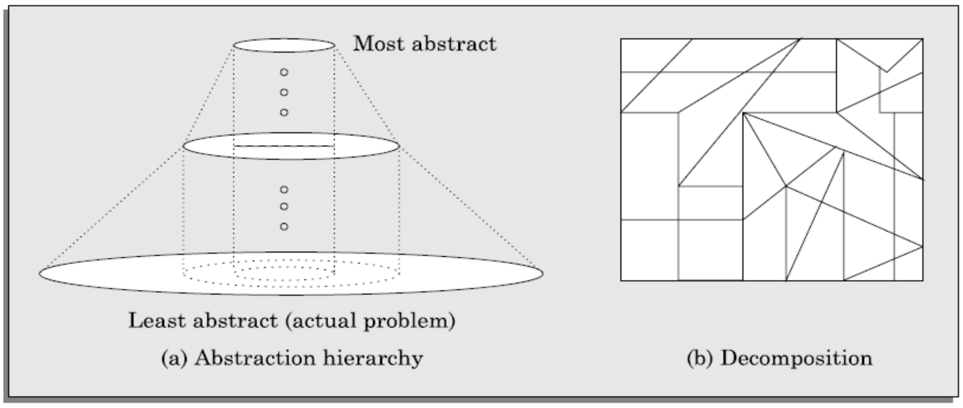


FIGURE 1.7 Schematic representation.

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.

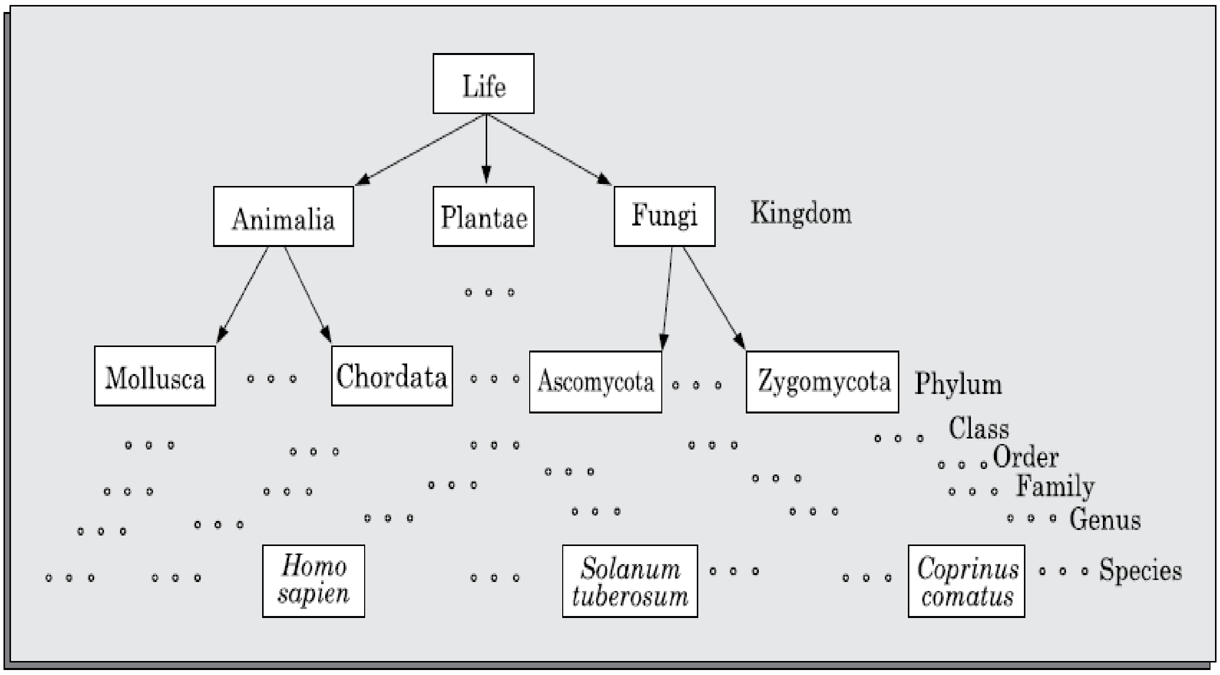


FIGURE 1.8 An abstraction hierarchy classifying living organisms.

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.

2. High-level Language Programming

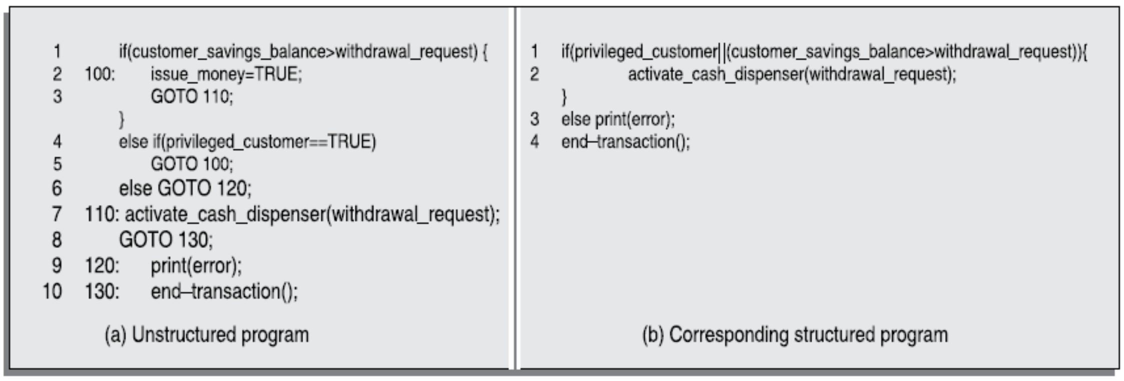
Computers became faster with the introduction of the semiconductor technology in the early 1960s. Faster semiconductor transistors replaced the prevalent vacuum tube-based circuits in a computer. With the availability of more powerful computers, it became possible to solve larger and more complex problems. 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.

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.

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).



```
      if(customer_savings_balance>withdrawal_request) {
100:      issue_money=TRUE;
          GOTO 110;
      }
      else if(privileged_customer==TRUE)
          GOTO 100;
      else GOTO 120;
110: activate_cash_dispenser(withdrawal_request);
      GOTO 130;
120:      print(error);
130:      end–transaction();
```

(a) Unstructured program

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

(b) Corresponding structured program

FIGURE 1.9 An example of (a) Unstructured program (b) Corresponding structured program.

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.

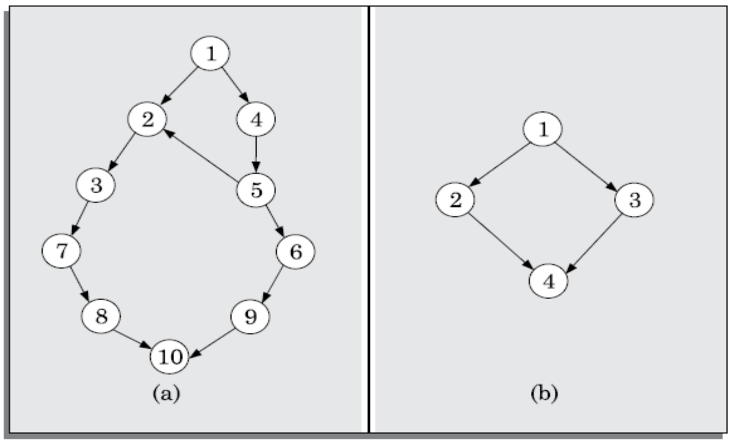


FIGURE 1.10 Control flow graphs of the programs of Figures 1.9(a) and (b).

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 1.10(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.

<u>Are GO TO statements the culprits?</u>

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.

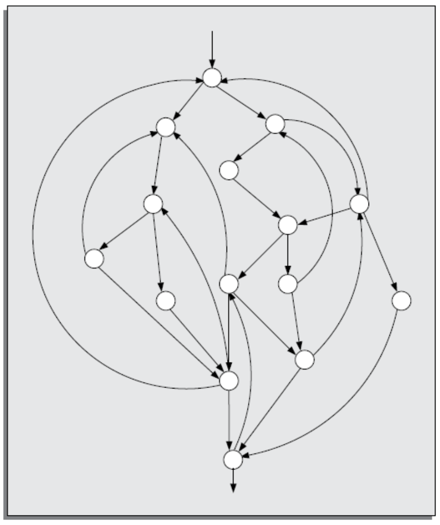


FIGURE 1.11 CFG of a program having too many GO TO statements.

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.

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 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.

5. <u>Data Flow-oriented Design</u>

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.

<u>DFDs: A crucial program representation for procedural program design</u>

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.

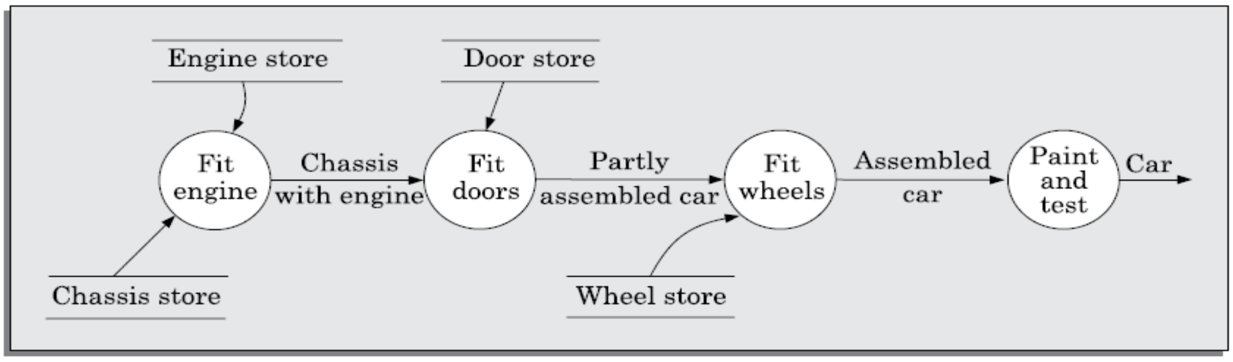


FIGURE 1.12 Data flow model of a car assembly plant.

6. 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.

UNIT-I

CHAPTER-II

SOFTWARE LIFE CYCLE MODELS

BASIC CONCEPTS

It is well known that all living organisms undergo a life cycle. For example when a seed is planted, it germinates, grows into a full tree, and finally dies. Based on this concept of a biological life cycle, the term software life cycle has been defined to imply the different stages (or phases) over which a software evolves from the initial customer request for it, to a fully developed software, and finally to a stage where it is no longer useful to any user, and is discarded.

As we have already pointed out, the life cycle of every software starts with a request expressing the need of the software by one or more customers. At this stage, the customers are usually not clear about all the features that would be needed, neither can they completely describe the identified features in concrete terms, and can only vaguely describe what is needed. This stage where the customer feels a need for the software and forms rough ideas about the required features is known as the inception stage. Starting with the inception stage, software evolves through a series of identifiable stages (also called phases) on account of the development activities carried out by the developers, until it is fully developed and is released to the customers.

Once installed and made available for use, the users start to use the software. This signals the start of the operation (also called maintenance) phase. As the users use the software, not only do they request for fixing any failures that they might encounter, but they also continually suggest several improvements and modifications to the software. Thus, the maintenance phase usually involves continually making changes to the software to accommodate the bug-fix and change requests from the user. The operation phase is usually the longest of all phases and constitutes the useful life of software. Finally the software is retired, when the users do not find it any longer useful due to reasons such as changed business scenario, availability of some new software having improved features and more efficient working, changed computing platforms, etc. This forms the essence of the life cycle of every software.

With this knowledge of a software life cycle, we discuss the concept of a software life cycle model and explore why it is necessary to follow a life cycle model in professional software development environments.

Software development life cycle (SDLC) model

A software development life cycle (SDLC) model (also called software life cycle model and software development process model) describes the different activities that need to be carried out for the software to evolve in its life cycle. Throughout our discussion, we shall use the terms software development life cycle (SDLC) and software development process interchangeably.
However, some authors distinguish an SDLC from a software development process. In their usage, a software development process describes the life cycle activities more precisely and elaborately, as compared to an SDLC. Also, a development process may not only describe various activities that are carried out over the life cycle, but also prescribe a specific methodology to carry out the activities,

and also recommends the specific documents and other artifacts that should be produced at the end of each phase.

SDLC can be considered to be a more generic term, as compared to the development process and several development processes may fit the same SDLC. An SDLC is represented graphically by drawing various stages of the life cycle and showing on it the transitions among the stages. This graphical model is usually accompanied by a textual description of various activities that need to be carried out during a phase before that phase can be considered to be complete.

Process *versus* methodology

A software development process has a much broader scope as compared to a software development methodology. A process usually describes all the activities starting from the inception of a software to its maintenance and retirement stages, or at least a chunk of activities in the life cycle. It also recommends specific methodologies for carrying out each activity. A methodology, in contrast, describes the steps to carry out only a single or at best a few individual activities.

Why use a development process?

Software development organisations have realised that adherence to a suitable life cycle model helps to produce good quality software and also helps to minimise the chances of time and cost overruns.

Programming-in-the-small refers to development of a toy program by a single programmer.
On the other hand, programming-in-the-large refers to development of professional software through team effort. While development of software of the former type could succeed even when an individual programmer uses a build and fix style of development, use of a suitable SDLC is essential for a professional software development project involving team effort to succeed.

Why document a development process?

A documented development process forms a common understanding of the activities to be carried out among the software developers and helps them to develop software in a systematic and disciplined manner. A documented development process model, besides preventing the misinterpretations that might occur when the development process is not adequately documented, also helps to identify inconsistencies, redundancies, and omissions in the development process.

Phase entry and exit criteria

The phase entry (or exit) criteria are usually expressed as a set of conditions that needs to be satisfied for the phase to start (or to complete). As an example, the phase exit criteria for the software requirements specification phase, can be that the software requirements specification (SRS) document is ready, has been reviewed internally, and also has been reviewed and approved by the customer. Only after these criteria are satisfied, the next phase can start.

If the entry and exit criteria for various phases are not well-defined, then that would leave enough scope for ambiguity in starting and ending various phases, and cause a lot of confusion among the developers. The decision regarding whether a phase is complete or not becomes subjective and it becomes difficult for the project manager to accurately tell how much has the development progressed.

When the phase entry and exit criteria are not well-defined, the developers might close the activities of a phase much before they are actually complete, giving a false impression of rapid progress.

WATERFALL MODEL AND ITS EXTENSIONS

The waterfall model and its derivatives were extremely popular in the 1970s and still are heavily being used across many development projects. The waterfall model is possibly the most obvious and intuitive way in which software can be developed through team effort. We can think of the waterfall model as a generic model that has been extended in many ways for catering to specific software development situations.

Classical Waterfall Model

Classical waterfall model is intuitively the most obvious way to develop software. It is simple but idealistic. In fact, it is hard to put this model into use in any non-trivial software development project, since developers do commit many mistakes during various development activities many of which are noticed only during a later phase. This requires revisiting the work of a previous phase to correct the mistakes, but the classical waterfall model has no provision to go back to modify the artifacts produced during an earlier phase.

The classical waterfall model divides the life cycle into six phases as shown in Figure 2.1.
It can be easily observed from this figure that the diagrammatic representation of the classical waterfall model resembles a multi-level waterfall. This resemblance justifies the name of the
Model

Phases of the classical waterfall model

The different phases of the classical waterfall model have been shown in Figure 2.1. As shown in Figure 2.1, the different phases are—feasibility study, requirements analysis and specification, design, coding and unit testing, integration and system testing, and maintenance. The phases starting from the feasibility study to the integration and system testing phase are known as the development phases. Software is developed during the development phases, and at the completion of the development phases, the software is delivered to the customer. After the delivery of software, customers start to use the software signalling the commencement of the operation phase. As the customers start to use the software, changes to it become necessary on account of bug fixes and feature extensions, causing maintenance works to be undertaken.

Therefore, the last phase is also known as the maintenance phase of the life cycle. It needs to be kept in mind that some of the text books have different number and names of the phases. An activity that spans all phases of software development is project management. Since it spans the entire project duration, no specific phase is named after it.

Project management, nevertheless, is an important activity in the life cycle and deals with managing all the activities that take place during software development and maintenance. In the waterfall model, different life cycle phases typically require relatively different amounts of efforts to be put in by the development team. The relative amounts of effort spent on different phases for typical software has been shown in Figure 2.2. Observe from Figure 2.2 that among all the life cycle phases, the maintenance phase normally requires the maximum effort. On the average, about 60 per cent of the total effort put in by the development team in the entire life cycle is spent on the maintenance activities alone. However, among the development phases, the integration and system testing phase requires the maximum effort in a typical development project. In the following subsection, we briefly describe the activities that are carried out in the different phases of the classical waterfall model.

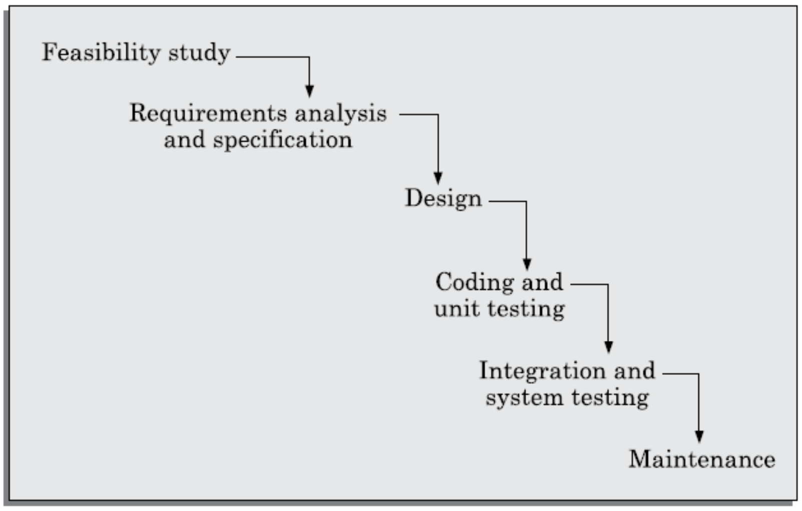

Feasibility study
Requirements analysis and specification
Design
Coding and unit testing
Integration and system testing
Maintenance

FIGURE 2.1 Classical waterfall model.

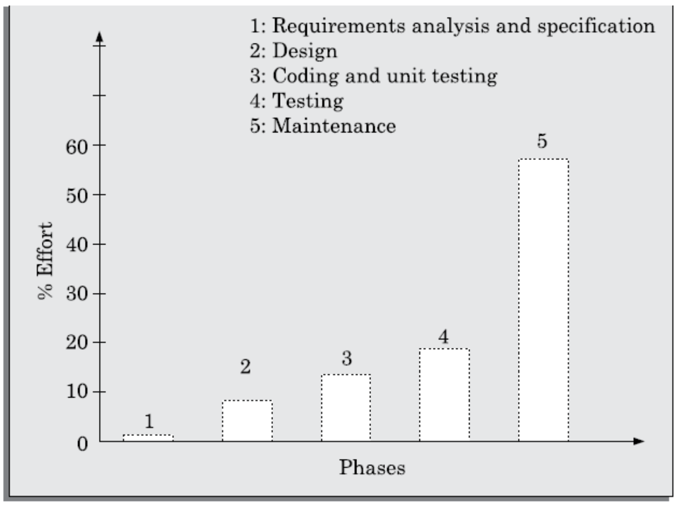


FIGURE 2.2 Relative effort distribution among different phases of a typical product.

Feasibility study

The main focus of the feasibility study stage is to determine whether it would be financially and technically feasible to develop the software. The feasibility study involves carrying out several activities such as collection of basic information relating to the software such as the different data items that would be input to the system, the processing required to be carried out on these data, the output data required to be produced by the system, as well as various constraints on the development. These collected data are analysed to perform the following:

Development of an overall understanding of the problem:

It is necessary to first develop an overall understanding of what the customer requires to be developed. For this, only the important requirements of the customer need to be understood and the details of various requirements such as the screen layouts required in the graphical user interface (GUI), specific formulas or algorithms required for producing the required results, and the databases schema to be used are ignored.

Formulation of various possible strategies for solving the problem: In this activity, various possible high-level solution schemes to the problem are determined. For example, solution in a client-server framework and a standalone application framework may be explored.

Evaluation of the different solution strategies: The different identified solution schemes are Analysed to evaluate their benefits and shortcomings. Such evaluation often requires making approximate estimates of the resources required, cost of development, and development time required.

Requirements analysis and specification

The aim of the requirements analysis and specification phase is to understand the exact requirements of the customer and to document them properly. This phase consists of two distinct activities, namely requirements gathering and analysis, and requirements specification. In the following, we give an overview of these two activities:

Requirements gathering and analysis: The goal of the requirements gathering activity is to collect all relevant information regarding the software to be developed from the customer with a view to clearly understand the requirements. For this, first requirements are gathered from the customer and then the gathered requirements are analysed. The goal of the requirements analysis activity is to weed out the incompleteness and inconsistencies in these gathered requirements. Note that an inconsistent requirement is one in which some part of the requirement contradicts some other part. On the other hand, an incomplete requirement is one in which some parts of the actual requirements have been omitted.
,,.
Requirements specification: After the requirement gathering and analysis activities are complete, the identified requirements are documented. This document is called a software requirements specification (SRS) document. The SRS document is written using end-user terminology. This makes the SRS document understandable to the customer. Therefore, understand ability of the SRS document is an important issue.

The SRS document normally serves as a contract between the development team and the customer. Any future dispute between the customer and the developers can be settled by examining the SRS document. The SRS document is therefore an important document which must be thoroughly understood by the development team, and reviewed jointly with the customer. The SRS document not only forms the basis for carrying out all the development activities, but several documents such as users' manuals, system test plan, etc. are prepared directly based on it.

Design

The goal of the design phase is to transform the requirements specified in the SRS document into a structure that is suitable for implementation in some programming language. In technical terms, during the design phase the software architecture is derived from the SRS document. Two distinctly different design approaches are popularly being used at present— the procedural and object-oriented design approaches.

Procedural design approach: The traditional procedural design approach is in use in many software development projects at the present time. This traditional design technique is based on data flow modelling. It consists of two important activities; first structured analysis of the requirements specification is carried out where the data flow structure of the problem is examined and modelled.

This is followed by a structured design step where the results of structured analysis are transformed into the software design.

During structured analysis, the functional requirements specified in the SRS document are decomposed into sub functions and the data-flow among these sub functions is analysed and represented diagrammatically in the form of DFDs.

The DFD technique is discussed in Chapter 6. Structured design is undertaken once the structured analysis activity is complete. Structured design consists of two main activities—architectural design (also called high-level design) and detailed design (also called Low-level design). High-level design involves decomposing the system into modules, and representing the interfaces and the invocation relationships among the modules. A high-level software design is sometimes referred to as the software architecture. During the detailed design activity, internals of the individual modules such as the data structures and algorithms of the modules are designed and documented.

Object-oriented design approach: In this technique, various objects that occur in the problem domain and the solution domain are first identified and the different relationships that exist among these objects are identified. The object structure is further refined to obtain the detailed design. The OOD approach is credited to have several benefits such as lower development time and effort, and better maintainability of the software.

Coding and unit testing

The purpose of the coding and unit testing phase is to translate a software design into source code and to ensure that individually each function is working correctly. The coding phase is also sometimes called the implementation phase, since the design is implemented into a workable solution in this phase. Each component of the design is implemented as a program module. The end-product of this phase is a set of program modules that have been individually unit tested. The main objective of unit testing is to determine the correct working of the individual modules. The specific activities carried out during unit testing include designing test cases, testing, debugging to fix problems, and management of test cases.

Integration and system testing

Integration of different modules is undertaken soon after they have been coded and unit tested. During the integration and system testing phase, the different modules are integrated in a planned manner. Various modules making up software are almost never integrated in one shot (can you guess the reason for this?). Integration of various modules is normally carried out incrementally over a number of steps. During each integration step, previously planned modules are added to the partially integrated system and the resultant system is tested. Finally, after all the modules have been successfully integrated and tested, the full working system is obtained. System testing is carried out on this fully working system

System testing usually consists of three different kinds of testing activities:
„ ***a*-testing:** *a* testing is the system testing performed by the development team.
„ ***b*-testing:** This is the system testing performed by a friendly set of customers.
„**Acceptance testing:** After the software has been delivered, the customer performs system testing to determine whether to accept the delivered software or to reject it.

Maintenance

The total effort spent on maintenance of a typical software during its operation phase is usually far greater than that required for developing the software itself. Many studies carried out in the past confirm this and indicate that the ratio of relative effort of developing a typical software product and the total effort spent on its maintenance is roughly 40:60.
Maintenance is required in the following three types of situations:

„. **Corrective maintenance:** This type of maintenance is carried out to correct errors that were not discovered during the product development phase.
„. **Perfective maintenance:** This type of maintenance is carried out to improve the performance of the system, or to enhance the functionalities of the system based on customer's requests.
„. **Adaptive maintenance:** Adaptive maintenance is usually required for porting the software to work in a new environment. For example, porting may be required to get the software to work on a new computer platform or with a new operating system.

Shortcomings of the classical waterfall model

The classical waterfall model is a very simple and intuitive model. However, it suffers from several shortcomings. Let us identify some of the important shortcomings of the classical waterfall model:

No feedback paths: In classical waterfall model, the evolution of software from one phase to the next is analogous to a waterfall. Just as water in a waterfall after having flowed down cannot flow back, once a phase is complete, the activities carried out in it and any artifacts produced in this phase are considered to be final and are closed for any rework.

This requires that all activities during a phase are flawlessly carried out. Contrary to a fundamental assumption made by the classical waterfall model, in practical development environments, the developers do commit a large number of errors in almost every activity they carry out during various phases of the life cycle. After all, programmers are humans and as the old adage says to err are human. The cause for errors can be many—oversight, wrong interpretations, use of incorrect solution scheme, communication gap, etc.

These defects usually get detected much later in the life cycle. For example, a design defect might go unnoticed till the coding or testing phase. Once a defect is detected at a later time, the developers need to redo some of the work done during that phase and also redo the work of later phases that are affected by the rework. Therefore, in any non-trivial software development project, it becomes nearly impossible to strictly follow the classical waterfall model of software development.

Difficult to accommodate change requests: This model assumes that all customer requirements can be completely and correctly defined at the beginning of the project. There is much emphasis on creating an unambiguous and complete set of requirements. But, it is hard to achieve this even in ideal project scenarios. The customers' requirements usually keep on changing with time. But, in this model it becomes difficult to accommodate any requirement change requests made by the customer after the requirements specification phase is complete, and this often becomes a source of customer discontent.

Inefficient error corrections: This model defers integration of code and testing tasks until it is very late when the problems are harder to resolve.

No overlapping of phases: This model recommends that the phases be carried out sequentially—new phase can start only after the previous one completes. However, it is rarely possible to adhere to this recommendation and it leads to a large number of team members to idle for extended periods. For example, for efficient utilisation of manpower, the testing team might need to design the system test cases immediately after requirements specification is complete

Is the classical waterfall model useful at all?

Irrespective of the life cycle model that is actually followed for a product development, the final documents are always written to reflect a classical waterfall model of development, so that comprehension of the documents becomes easier for any one reading the document.

Iterative Waterfall Model

We had pointed out in the previous section that in a practical software development project, the classical waterfall model is hard to use. We had branded the classical waterfall model as an idealistic model. In this context, the iterative waterfall model can be thought of as incorporating the necessary changes to the classical waterfall model to make it usable in practical software development projects.

The feedback paths introduced by the iterative waterfall model are shown in Figure 2.3.
The feedback paths allow for correcting errors committed by a programmer during some phase, as and when these are detected in a later phase. For example, if during the testing phase a design error is identified, then the feedback path allows the design to be reworked and the changes to be reflected in the design documents and all other subsequent documents. Please notice that in Figure 2.3 there is no feedback path to the feasibility stage. This is because once a team accepts to take up a project; it does not give up the project easily due to legal and moral reasons.

Almost every life cycle model that we discuss is iterative in nature, except the classical waterfall model and the V-model—which are sequential in nature. In a sequential model, once a phase is complete, no work product of that phase is changed later.

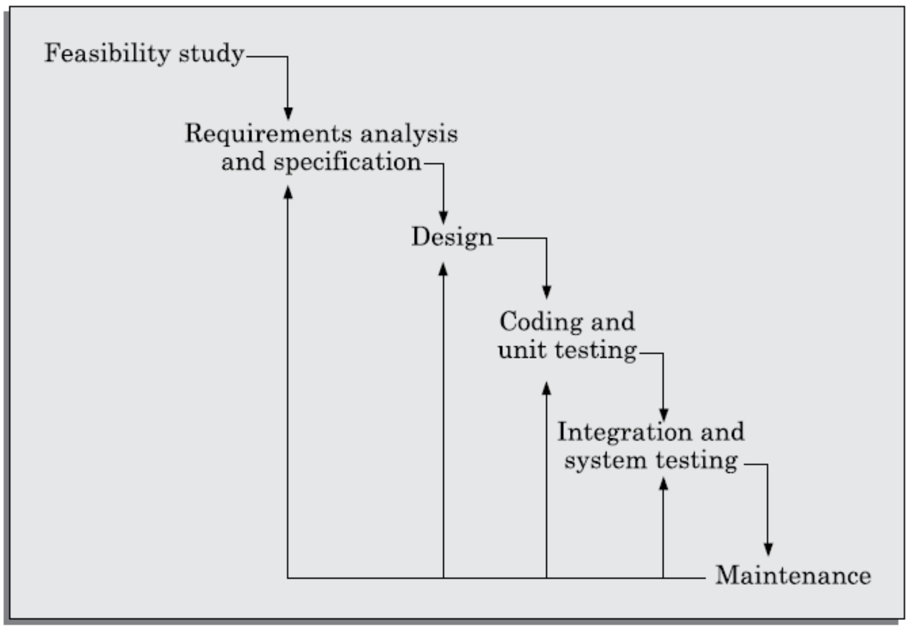

Feasibility study
Requirements analysis and specification
Design
Coding and unit testing
Integration and system testing
Maintenance

FIGURE 2.3 Iterative waterfall model.

Phase containment of errors

No matter how careful a programmer may be, he might end up committing some mistake or other while carrying out a life cycle activity. These mistakes result in errors (also called faults or bugs) in the work product. It is advantageous to detect these errors in the same phase in which they take place, since early detection of bugs reduces the effort and time required for correcting those. For example, if a design problem is detected in the design
phase itself, then the problem can be taken care of much more easily than if the error is identified, say, at the end of the testing phase. In the later case, it would be necessary not only to rework the design, but also to appropriately redo the relevant coding as well as the testing activities, thereby incurring higher cost. It may not always be possible to detect all the errors in the same phase in which they are made. Nevertheless, the errors should be detected as early as possible.

For achieving phase containment of errors, how can the developers detect almost all error that they commit in the same phase? After all, the end product of many phases is text or graphical documents, e.g. SRS document, design document, test plan document, etc.
A popular technique is to rigorously review the documents produced at the end of a phase

Phase overlap

Even though the strict waterfall model envisages sharp transitions to occur from one phase to the next (see Figure 2.3), in practice the activities of different phases overlap (as shown in Figure 2.4) due to two main reasons:
„. In spite of the best effort to detect errors in the same phase in which they are committed, some errors escape detection and are detected in a later phase. These subsequently detected errors cause the activities of some already completed phases to be reworked. If we consider such rework after a phase is complete, we can say that the activities pertaining to a phase do not end at the completion of the phase, but overlap with other phases as shown in Figure 2.4.

An important reason for phase overlap is that usually the work required to be carried out in a phase is divided among the team members. Some members may complete their part of the work earlier than other members. If strict phase transitions are maintained, then the team members who complete their work early would idle waiting for the phase to be complete, and are said to be in a blocking state. Thus the developers who complete early would idle while waiting for their team mates to complete their assigned work. Clearly this is a cause for wastage of resources and a source of cost escalation and inefficiency. As a result, in real projects, the phases are allowed to overlap. That is, once a developer completes his work assignment for a phase, proceeds to start the work for the next phase, without waiting for all his team members to complete their respective work allocations.

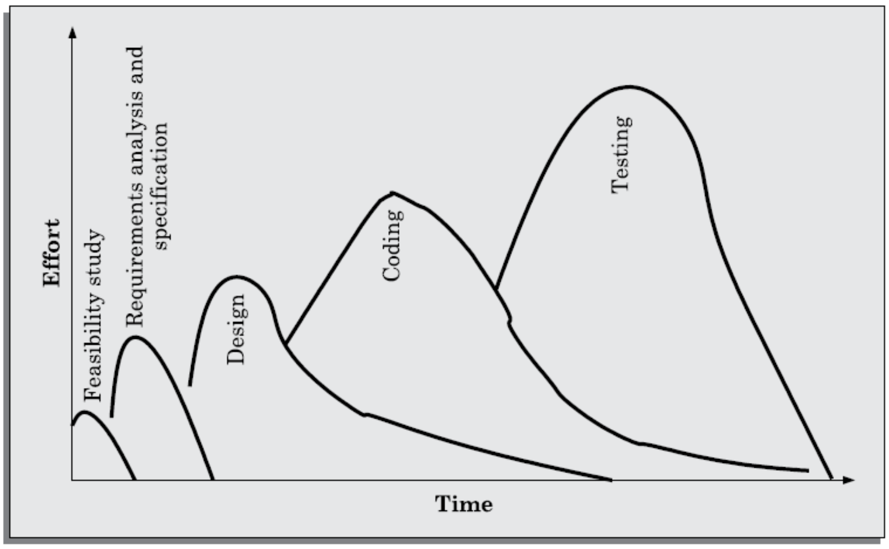


FIGURE 2.4 Distribution of effort for various phases in the iterative waterfall model.

Shortcomings of the iterative waterfall model

The iterative waterfall model is a simple and intuitive software development model. It was used satisfactorily during 1970s and 1980s. However, the characteristics of software development projects have changed drastically over years. In the 1970s and 1960s, software development projects spanned several years and mostly involved generic software product development. The projects are now shorter, and involve Customised software development.

Further, software was earlier developed from scratch. Now the emphasis is on as much reuse of code and other project artifacts as possible. Waterfall-based models have worked satisfactorily over last many years in the past. The situation has changed substantially now.

As pointed out in the first chapter several decades back, every software was developed from scratch. Now, not only software has become very large and complex, very few (if at all any) software project is being developed from scratch. The software services (customised software) are poised to become the dominant types of projects. In the present software development projects, use of waterfall model causes several problems. In this context, the agile models have been proposed about a decade back that attempt to overcome the important shortcomings of the waterfall model by suggesting certain radical modifications to the waterfall style of software development.

Difficult to accommodate change requests: Once requirements have been frozen, the waterfall model provides no scope for any modifications to the requirements. The basic assumption made in the iterative waterfall model that methodical requirements gathering and analysis alone is sufficient to comprehensively and correctly identify all the requirements by the end of the requirements phase is flawed.

Incremental delivery not supported: In the iterative waterfall model, the full software is completely developed and tested before it is delivered to the customer. There is no provision for any intermediate deliveries to occur. This is problematic because the complete application may take several months or years to be completed and delivered to the customer. By the time the software is delivered, installed, and becomes ready for use, the customer's business process might have changed substantially. This makes the developed application a poor fit to the customer's requirements.

Phase overlap not supported: For most real-life projects, it becomes difficult to follow the rigid phase sequence prescribed by the waterfall model. By the term a rigid phase sequence, we mean that a phase can start only after the previous phase is complete in all respects.
As already discussed, strict adherence to the waterfall model creates blocking states. That is, if some team members complete their assigned work for a phase earlier than other team members, would have to ideal, and wait for the others to complete their work before initiating the work for the next phase. The waterfall model is usually adapted for use in real-life projects by allowing overlapping of various phases as shown in Figure 2.4.

Error correction unduly expensive: In waterfall model, validation is delayed till the complete development of the software. As a result, the defects that are noticed at the time of validation incur expensive rework and result in cost escalation and delayed delivery.

Limited customer interactions: This model supports very limited customer interactions. It is generally accepted that software developed in isolation from the customer is the cause of many problems. In fact, interactions occur only at the start of the project and at project completion. As a result, the developed software usually turns out to be a misfit to the customer's actual requirements.

Heavy weight: The waterfall model overemphasises documentation. A significant portion of the time of the developers is spent in preparing documents, and revising them as changes occur over the life cycle. Heavy documentation though useful during maintenance and for carrying out review, is a source of team inefficiency.

No support for risk handling and reuse: It becomes difficult to use the waterfall model in projects that are susceptible to various types of risks, or those involving significant reuse of existing development artifacts. Please recollect that software services types of projects usually involve significant reuse of requirements, design and code. Therefore, it becomes difficult to use waterfall model in services type of projects.

V-Model

A popular development process model, V-model is a variant of the waterfall model. As is the case with the waterfall model, this model gets its name from its visual appearance (see Figure 2.5). In this model verification and validation activities are carried out throughout the development life cycle, and therefore the chances bugs in the work products considerably reduce. This model is therefore generally considered to be suitable for use in projects concerned with development of safety-critical software that are required to have high reliability.

As shown in Figure 2.5, there are two main phases—development and validation phases. The left half of the model comprises the development phases and the right half comprises the validation phases.

In each development phase, along with the development of a work product, test case design and the plan for testing the work product are carried out, whereas the actual testing is carried out in the validation phase. This validation plan created during the development phases is carried out in the corresponding validation phase which has been shown by dotted arcs in Figure 2.5.

In the validation phase, testing is carried out in three steps—unit, integration, and system testing. The purpose of these three different steps of testing during the validation phase is to detect defects that arise in the corresponding phases of software development— requirements analysis and specification, design, and coding respectively.

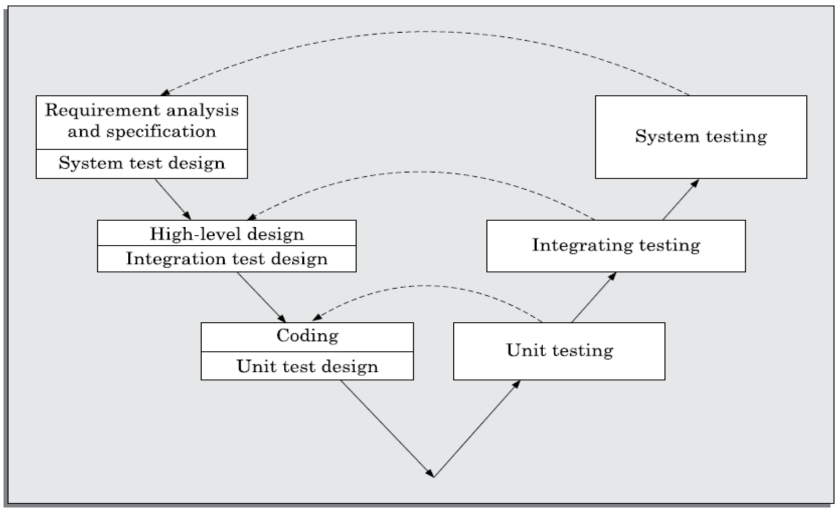


FIGURE 2.5 V-model.

V-model *versus* waterfall model

We have already pointed out that the V-model can be considered to be an extension of the waterfall model. However, there are major differences between the two. As already mentioned, in contrast to the iterative waterfall model where testing activities are confined to the testing phase only, in the V-model testing activities are spread over the entire life cycle. As shown in Figure 2.5, during the requirements specification phase, the system test suite design activity takes place. During the design phase, the integration test cases are designed. During coding, the unit test cases are designed. Thus, we can say that in this model, development and validation activities proceed hand in hand.

Advantages of V-model

The important advantages of the V-model over the iterative waterfall model are as following:

- „. In the V-model, much of the testing activities (test case design, test planning, etc.) are carried out in parallel with the development activities. Therefore, before testing phase starts significant part of the testing activities, including test case design and test planning, is already complete. Therefore, this model usually leads to a shorter testing phase and an overall faster product development as compared to the iterative model.
- Since test cases are designed when the schedule pressure has not built up, the quality of the test cases are usually better.
- The test team is reasonably kept occupied throughout the development cycle in contrast to the waterfall model where the testers are active only during the testing phase. This leads to more efficient manpower utilisation
- In the V-model, the test team is associated with the project from the beginning. Therefore they build up a good understanding of the development artifacts, and this in turn, helps them to carry out effective testing of the software. In contrast, in the waterfall model often the test team comes on board late in the development cycle, since no testing activities are carried out before the start of the implementation and testing phase.

Disadvantages of V-model

Being a derivative of the classical waterfall model, this model inherits most of the weaknesses of the waterfall model.

Prototyping Model

The prototype model is also a popular life cycle model. The prototyping model can be considered to be an extension of the waterfall model. This model suggests building a working prototype of the system, before development of the actual software. A prototype is a toy and crude implementation of a system. It has limited functional capabilities, low reliability, or inefficient performance as compared to the actual software. A prototype can be built very quickly by using several shortcuts. The shortcuts usually involve developing inefficient, inaccurate, or dummy functions. The shortcut implementation of a function.

for example, may produce the desired results by using a table look-up rather than by performing the actual computations. Normally the term rapid prototyping is used when software tools are used for prototype construction. For example, tools based on fourth generation languages (4GL) may be used to construct the prototype for the GUI parts.

Necessity of the prototyping model

The prototyping model is advantageous to use for specific types of projects. In the following, we identify three types of projects for which the prototyping model can be followed to advantage:

It is advantageous to use the prototyping model for development of the graphical user interface (GUI) part of an application. Through the use of a prototype, it becomes easier to illustrate the input data formats, messages, reports, and the interactive dialogs to the customer.
This is a valuable mechanism for gaining better understanding of the customers' needs. In this regard, the prototype model turns out to be especially useful in developing the graphical user interface (GUI) part of a system. For the user, it becomes much easier to form an opinion regarding what would be more suitable by experimenting with a working user interface, rather than trying to imagine the working of a hypothetical user interface.

The prototyping model is especially useful when the exact technical solutions are unclear to the development team. A prototype can help them to critically examine the technical issues associated with product development. For example, consider a situation where the development team has to write a command language interpreter as part of a graphical user interface development. Suppose none of the team members has ever written a compiler before. Then, this lack of familiarity with a required development technology is a technical risk. This risk can be resolved by developing a prototype compiler for a very small language to understand the issues associated with writing a compiler for a command language. Once they feel confident in writing compiler for the small language, they can use this knowledge to develop the compiler for the command language. Often, major design decisions depend on issues such as the response time of a hardware controller, or the efficiency of a sorting algorithm, etc. In such circumstances, a prototype is often the best way to resolve the technical issues.

An important reason for developing a prototype is that it is impossible to "get it right" the first time. As advocated by Brooks [1975], one must plan to throw away the prototype software in order to develop a good software later. Thus, the prototyping model can be deployed when development of highly optimised and efficient software is required.

Life cycle activities of prototyping model

The prototyping model of software development is graphically shown in Figure 2.6. As shown in Figure 2.6, software is developed through two major activities—prototype construction and iterative waterfall-based software development.

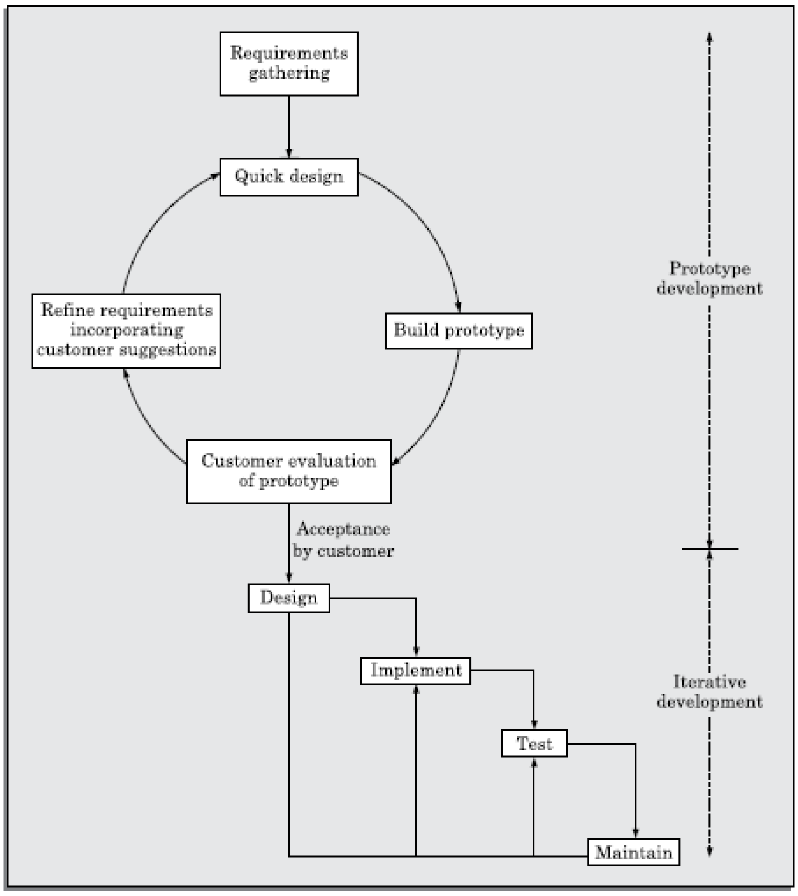


FIGURE 2.6 Prototyping model of software development.

Prototype development: Prototype development starts with an initial requirements gathering phase. A quick design is carried out and a prototype is built. The developed prototype is submitted to the customer for evaluation. Based on the customer feedback, the requirements are refined and the prototype is suitably modified. This cycle of obtaining customer feedback and modifying the prototype continues till the customer approves the prototype.

Iterative development: Once the customer approves the prototype, the actual software is developed using the iterative waterfall approach. In spite of the availability of a working prototype, the SRS document is usually needed to be developed since the SRS document is invaluable for carrying out traceability analysis, verification, and test case design during later phases. However, for GUI parts, the requirements analysis and specification phase becomes redundant since the working prototype that has been approved by the customer serves as an animated requirements specification.

The code for the prototype is usually thrown away. However, the experience gathered from developing the prototype helps a great deal in developing the actual system. By constructing the prototype and submitting it for user evaluation, many customer requirements get properly defined and technical issues get resolved by experimenting with the prototype. This minimises later change requests from the customer and the associated redesign costs.

Strengths of the prototyping model

This model is the most appropriate for projects that suffer from risks arising from technical uncertainties and unclear requirements. A constructed prototype helps overcome these risks.

Weaknesses of the prototyping model

The prototype model can increase the cost of development for projects that are routine development work and do not suffer from any significant risks. Even when a project is susceptible to risks, the prototyping model is effective only for those projects for which the risks can be identified upfront before the development starts. Since the prototype is constructed only at the start of the project, the prototyping model is ineffective for risks identified later during the development cycle. The prototyping model would not be appropriate for projects for which the risks can only be identified after the development is underway. For example, the risk of key personnel leaving the project midway is hard to predict at the start of the project.

Incremental Development Model

In the incremental life cycle model, the software is developed in increment. In this life cycle model, first the requirements are split into a set of increments. The first increment is a simple working system implementing only a few basic features. Over successive iterations, successive increments are implemented and delivered to the customer until the desired system is realised. The incremental development model has been shown in Figure 2.7.

Life cycle activities of incremental development model

In the incremental life cycle model, the requirements of the software are first broken down into several modules or features that can be incrementally constructed and delivered. This has been pictorially depicted in Figure 2.7. At any time, plan is made only for the next increment and no long-term plans are made. Therefore, it becomes easier to accommodate change requests from the customers.

The development team first undertakes to develop the core features of the system. The core or basic features are those that do not need to invoke any services from the other features. On the other hand, non-core features need services from the core features. Once the initial core features are developed, these are refined into increasing levels of capability by adding new functionalities in successive versions. Each incremental version is usually developed using an iterative waterfall model of development. The incremental model is schematically shown in Figure 2.8. As each successive version of the software is constructed and delivered to the customer, the customer feedback is obtained on the delivered version and these feedbacks are incorporated in the next version. Each delivered version of the software incorporates additional features over the previous version and also refines the features that were already delivered to the customer.
The incremental model has schematically been shown in Figure 2.8.

After the requirements gathering and specification, the requirements are split into several increments. Starting with the core (increment 1), in each successive iteration, the next increment is constructed using a waterfall model of development and deployed at the customer site.

After the last (shown as increment *n*) has been developed and deployed at the client site, the full software is developed and deployed.

Advantages

The incremental development model offers several advantages. Two important ones are the following:

Error reduction: The core modules are used by the customer from the beginning and therefore these get tested thoroughly. This reduces chances of errors in the core modules of the final product, leading to greater reliability of the software.

Incremental resource deployment: This model obviates the need for the customer to commit large resources at one go for development of the system. It also saves the developing organisation from deploying large resources and manpower for a project in one go.

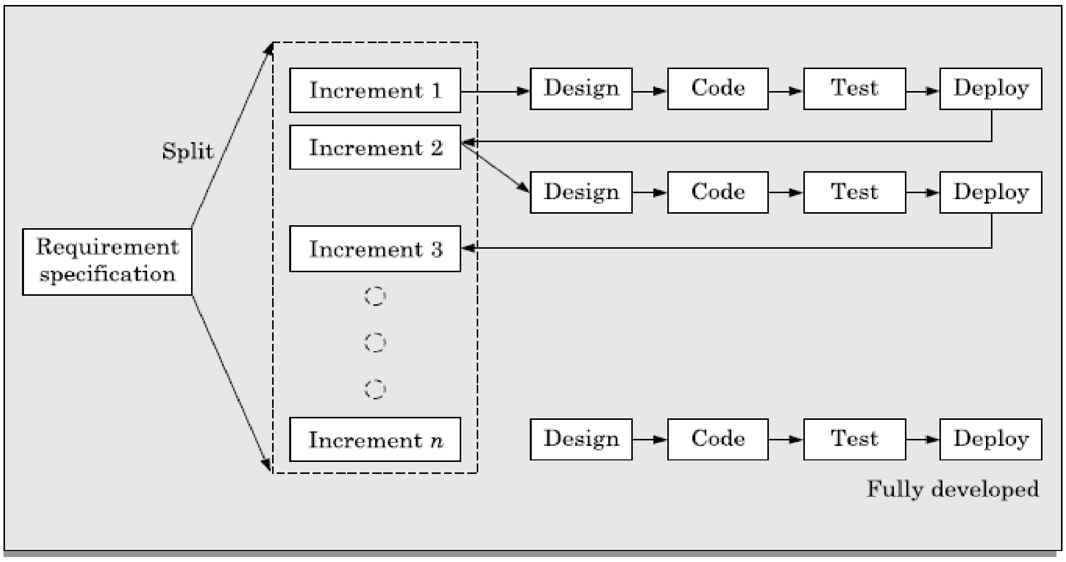


FIGURE 2.8 Incremental model of software development.

Evolutionary Model

This model has many of the features of the incremental model. As in case of the incremental model, the software is developed over a number of increments. At each increment, a concept (feature) is implemented and is deployed at the client site. The software is successively refined and feature-enriched until the full software is realised. The principal idea behind the evolutionary life cycle model is conveyed by its name. In the incremental development model, complete requirements are first developed and the SRS document prepared. In contrast, in the evolutionary model, the requirements, plan, estimates, and solution evolve over the iterations, rather than fully defined and frozen in a major up-front specification effort before the development iterations begin. Such evolution is consistent with the pattern of unpredictable feature discovery and feature changes that take place in new product development.

Though the evolutionary model can also be viewed as an extension of the waterfall model, but it incorporates a major paradigm shift that has been widely adopted in many recent life cycle models. Due to obvious reasons, the evolutionary software development process is sometimes referred to as design a little, build a little, test a little, and deploy a little model. This means that after the requirements have been specified, the design, build, test, and deployment activities are iterated. A schematic representation of the evolutionary model of development has been shown in Figure 2.9.

Advantages

The evolutionary model of development has several advantages. Two important advantages of using this model are the following:

„ „
Effective elicitation of actual customer requirements: In this model, the user gets a chance to experiment with partially developed software much before the complete requirements are developed. Therefore, the evolutionary model helps to accurately elicit user requirements with the help of feedback obtained on the delivery of different versions of the software. As a result, the change requests after delivery of the complete software gets substantially reduced.

„ „
Easy handling change requests: In this model, handling change requests is easier as no long-term plans are made. Consequently, reworks required due to change requests are normally much smaller compared to the sequential models.

Disadvantages

The main disadvantages of the successive versions model are as follows:

„ „
Feature division into incremental parts can be non-trivial: For many development projects, especially for small-sized projects, it is difficult to divide the required features into several parts that can be incrementally implemented and delivered. Further, even for larger problems, often the features are so intertwined and dependent on each other that even an expert would need considerable effort to plan the incremental deliveries.

„ **. Ad hoc design:** Since at a time design for only the current increment is done, the design can become ad hoc without specific attention being paid to maintainability and optimality. Obviously, for moderate sized problems and for those for which the customer requirements are clear, the iterative waterfall model can yield a better solution.

Evolutionary versus incremental model of developments

The evolutionary and incremental have several things in common, such as incremental development and deployment at the client site. However, in a purely incremental model, the requirement specification is completed before any development activities start. Once the requirement specification is completed, the requirements are split into requirements. In a purely evolutionary development, the development of the first version starts off after obtaining a rough understanding of what is required. As the development proceeds, more and more requirements emerge. The modern development models, such as the agile models are neither purely incremental, nor purely evolutionary, but are somewhat in between and are referred to as incremental and evolutionary model. In this mode, the initial requirements are obtained and specified, but requirements that emerge later are accommodated.

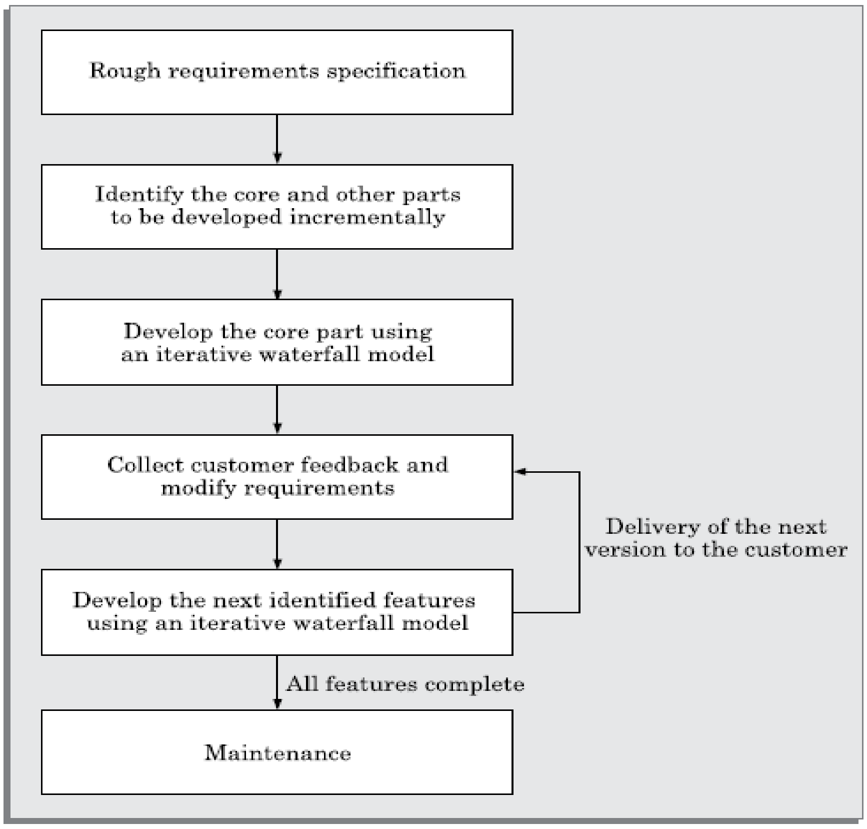

Rough requirements specification
Identify the core and other parts to be developed incrementally
Develop the core part using an iterative waterfall model
Collect customer feedback and modify requirements
Delivery of the next version to the customer
Develop the next identified features using an iterative waterfall model
All features complete
Maintenance

FIGURE 2.9 Evolutionary model of software development.

RAPID APPLICATION DEVELOPMENT (RAD)

The rapid application development (RAD) model was proposed in the early nineties in an attempt to overcome the rigidity of the waterfall model (and its derivatives) that makes it difficult to accommodate any change requests from the customer. It proposed a few radical extensions to the waterfall model. This model has the features of both prototyping and evolutionary models. It deploys an evolutionary delivery model to obtain and incorporate the customer feedbacks on incrementally delivered versions. In this model prototypes are constructed, and incrementally the features are developed and delivered to the customer. But unlike the prototyping model, the prototypes are not thrown away but are enhanced and used in the software construction.

Main motivation

In the iterative waterfall model, the customer requirements need to be gathered, analysed, documented, and signed off upfront, before any development could start. However, often clients do not know what they exactly wanted until they saw a working system. It has now become well accepted among the practitioners that only through the process commenting on an installed application that the exact requirements can be brought out. But in the iterative waterfall model, the customers do not get to see the software, until the development is complete in all respects and the software has been delivered and installed.

Naturally, the delivered software often does not meet the customer expectations and many change requests are generated by the customer. The changes are incorporated through subsequent maintenance efforts. This made the cost of accommodating the changes extremely high and it usually took a long time to have a good solution in place that could reasonably meet the requirements of the customers. The RAD model tries to overcome this problem by inviting

Working of RAD

In the RAD model, development takes place in a series of short cycles or iterations. At any time, the development team focuses on the present iteration only, and therefore plans are made for one increment at a time. The time planned for each iteration is called a time box.

Each iteration is planned to enhance the implemented functionality of the application by only a small amount. During each time box, quick-and-dirty prototype-style software for some functionality is developed. The customer evaluates the prototype and gives feedback on the specific improvements that may be necessary. The prototype is refined based on the customer feedback.

Please note that the prototype is not meant to be released to the customer for regular use though.
The development team almost always includes a customer representative to clarify the requirements. This is intended to make the system tuned to the exact customer requirements and also to bridge the communication gap between the customer and the development team. The development team usually consists of about five to six members, including a customer representative.

How does RAD facilitate accommodation of change requests?

The customers usually suggest changes to a specific feature only after they have used it. Since the features are delivered in small increments, the customers are able to give their change requests pertaining to a feature already delivered. Incorporation of such change requests just after the delivery of an incremental feature saves cost as this is carried out before large investments have been made in development and testing of a large number of features.

How does RAD facilitate faster development?

The decrease in development time and cost, and at the same time an increased flexibility to incorporate changes are achieved in the RAD model in two main ways—minimal use of planning and heavy reuse of any existing code through rapid prototyping. The lack of long-term and detailed planning gives the flexibility to accommodate later requirements changes. Reuse of existing code has been adopted as an important mechanism of reducing the development cost.

RAD model emphasises code reuse as an important means for completing a project faster. In fact, the adopters of the RAD model were the earliest to embrace object-oriented languages and practices. Further, RAD advocates use of specialised tools to facilitate fast creation of working prototypes. These specialised tools usually support the following features:

„ . Visual style of development.
„ . Use of reusable components.

Applicability of RAD Model

The following are some of the characteristics of an application that indicate its suitability to RAD style of development:

„ „ **Customised software:** As already pointed out a customised software is developed for one or two customers only by adapting an existing software. In customised software development projects, substantial reuse is usually made of code from pre-existing software. For example, a company might have developed a software for automating the data processing activities at one or more educational institutes. When any other institute requests for an automation package to be developed, typically only a few aspects needs to be tailored—since among different educational institutes, most of the data processing activities such as student registration, grading, fee collection, estate management, accounting, maintenance of staff service records, etc. are similar to a large extent. Projects involving such tailoring can be carried out speedily and cost-effectively using the RAD model.

„ „
Non-critical software: The RAD model suggests that quick and dirty software should first be developed and later this should be refined into the final software for delivery. Therefore, the developed product is usually far from being optimal in performance and reliability. In this regard, for well understood development projects and where the scope of reuse is rather restricted, the iterative waterfall model may provide a better solution.

„ „ **Highly constrained project schedule:** RAD aims to reduce development time at the expense of good documentation, performance, and reliability. Naturally, for projects with very aggressive time schedules, RAD model should be preferred.

„ „
Large software: Only for software supporting many features (large software) can incremental development and delivery be meaningfully carried out.

Application characteristics that render RAD unsuitable

The RAD style of development is not advisable if a development project has one or more of the following characteristics:
Generic products (wide distribution): software products are generic in nature and usually have wide distribution. For such systems, optimal performance and reliability are imperative in a competitive market. As it has already been discussed, the RAD model of development may not yield systems having optimal performance and reliability.
Requirement of optimal performance and/or reliability: For certain categories of products, optimal performance or reliability is required. Examples of such systems include an operating system (high reliability required) and a flight simulator software (high performance required). If such systems are to be developed using the RAD model, the desired product performance and reliability may not be realised.
Lack of similar products: If a company has not developed similar software, then it would hardly be able to reuse much of the existing artifacts. In the absence of sufficient plug-in components, it becomes difficult to develop rapid prototypes through reuse, and use of RAD model becomes meaningless.
Monolithic entity: For certain software, especially small-sized software, it may be hard to divide the required features into parts that can be incrementally developed and simulator software (high performance required). If such systems are to be developed using the RAD model, the desired product performance and reliability may not be realised.

Lack of similar products: If a company has not developed similar software, then it would hardly be able to reuse much of the existing artifacts. In the absence of sufficient plug-in components, it becomes difficult to develop rapid prototypes through reuse, and use of RAD model becomes meaningless.

Monolithic entity: For certain software, especially small-sized software, it may be hard to divide the required features into parts that can be incrementally developed and delivered. In this case, it becomes difficult to develop software incrementally.

Comparison of RAD with Other Models

In this section, we compare the relative advantages and disadvantages of RAD with other life cycle models.

RAD versus prototyping model

In the prototyping model, the developed prototype is primarily used by the development team to gain insights into the problem, choose between alternatives, and elicit customer feedback. The code developed during prototype construction is usually thrown away. In contrast, in RAD it is the developed prototype that evolves into the deliverable software.

RAD *versus* iterative waterfall model

In the iterative waterfall model, all the functionalities of software are developed together. On the other hand, in the RAD model the product functionalities are developed incrementally through heavy code and design reuse. Further, in the RAD model customer feedback is obtained on the developed prototype after each iteration and based on this the prototype is refined. Thus, it becomes easy to accommodate any request for requirements changes. However, the iterative waterfall model does not support any mechanism to accommodate any requirement change requests. The iterative waterfall model does have some important advantages that include the following. Use of the iterative waterfall model leads to production of good quality documentation which can help during software maintenance. Also, the developed software usually has better quality and reliability than that developed using RAD.

RAD *versus* evolutionary model

Incremental development is the hallmark of both evolutionary and RAD models. However, in RAD each increment results in essentially a quick and dirty prototype, whereas in the evolutionary model each increment is systematically developed using the iterative waterfall model. Also in the RAD model, software is developed in much shorter increments compared the evolutionary model. In other words, the incremental functionalities that are developed are of fairly larger granularity in the evolutionary model.

AGILE DEVELOPMENT MODELS

As already pointed out, though the iterative waterfall model has been very popular during the 1970s and 1980s, developers face several problems while using it on present-day software projects. The main difficulties include handling change requests from customers during product development, and the unreasonably high cost and time that is incurred while developing customised applications.

Capers Jones carried out research involving 800 real-life software development projects, and concluded that on the average 40 per cent of the requirements are arrived well after the development has already begun. In this context, over the last two decades or so, several life cycle models have been proposed to overcome some of the glaring shortcomings of the waterfall-based models that become conspicuous when used in modern software development projects. In the following, we identify a few reasons why the waterfall-based development was becoming difficult to use in project in recent times:

In the traditional iterative waterfall-based software development models, the requirements for the system are determined at the start of a development project and are assumed to be fixed from that point on. Later changes to the requirements after the SRS document has been completed are discouraged. If requirement changes become unavoidable later during the development life cycle, then the cost of accommodating these changes becomes prohibitively high. On the other hand, accumulated experience indicates that customers frequently change their requirements during the development period due to a variety of reasons. This is a major source of cost escalation in waterfall-based development projects.

over the last two decades or so, customised applications (services) have become common place. The prevalence of customised application development can be gauged from the fact that the sales revenue generated worldwide from services already exceeds that of the software products. Iterative waterfall model is not suitable for development of such software. The main reason for this is that customisation essentially involves reusing most of the parts of an existing application and carrying out only minor modifications to the other parts. For such development projects, the need for appropriate development models was acutely felt, and many researchers started to investigate this issue.

„ . Waterfall model is called a heavy weight model, since there is too much emphasis on producing documentation and usage of tools. This is often a source of inefficiency and causes the project completion time to be much longer in comparison to the customer expectations.

„ . Waterfall model prescribes almost no customer interactions after the requirements have been specified. In fact, in the waterfall model of software development, customer interactions are largely confined to the project initiation and project completion stages.

The agile software development model was proposed in the mid-1990s to overcome the shortcomings of the waterfall model of development identified above. The agile model could help a project to adapt to change requests quickly.1 Thus; a major aim of the agile models is to facilitate quick project completion. But, how is agility achieved in these models? Agility is achieved by fitting the process to the project. That is, it gave the required flexibility so that the activities that may not be necessary for a specific project could be easily removed. Also, anything that wastes time and effort is avoided.
Please note that agile model is being used as an umbrella term to refer to a group of development processes. While these processes share certain common characteristics, yet they do have certain subtle differences among themselves. A few popular agile SDLC models are the following:

„ . Crystal
„ . Atern (formerly DSDM)
„ . Feature-driven development
„ . Scrum
„ . Extreme programming (XP)
„ . Lean development
„ . Unified process

In an agile model, the requirements are decomposed into many small parts that can be incrementally developed. The agile models adopt an incremental and iterative approach. Each incremental part is developed over an iteration. Each iteration is intended to be small and easily manageable and lasts for a couple of weeks only. At a time, only one increment is planned, developed, and then deployed at the customer site. No long-term plans are made. The time to complete iteration is called a *time box*. The implication of the term *time box* is that the end date for iteration does not change. That is, the delivery date is considered sacrosanct. The development team can, however, decide to reduce the delivered functionality during a time box if necessary.
A central principle of the agile model is the delivery of an increment to the customer after each time box. A few other principles that are central to the agile model are discussed below.

Essential Idea behind Agile Models

For establishing close interactions with the customer during development and to gain a clear understanding of domain-specific issues, each agile project usually includes a customer representative in the team. At the end of each iteration, stakeholders and the customer representative review the progress made and re-evaluate the requirements. The developed increment is installed at the customer site.
Distinguishing characteristics of the agile models is frequent delivery of software increments to the customer. Agile models emphasise use of face-to-face communication in preference over written documents. It is recommended that the development team size be kept small (5-9 people). This would help the team members to meaningfully engage in face-to-face communication and lead to a collaborative work environment. It is implicit that the agile model is suited to the development of small projects. However, can agile development be used for large projects with geographically dispersed team? In this case, the agile model can be used with different teams maintaining as much

daily contact as possible through various communication media such as video conferencing, telephone, e-mail, etc.

The following important principles behind the agile model were publicised in the form of a manifesto in 2001:
„ . Working software over comprehensive documentation.
„ . Frequent delivery of incremental versions of the software to the customer in intervals of few weeks.
„ . Requirement change requests from the customer are encouraged and are to be efficiently incorporated.
„ . Having competent team members and enhancing interactions among them is considered much more important than issues such as usage of sophisticated tools or strict adherence to a documented process. It is advocated that enhanced communication among the development team members can be realised through face-to-face communication rather than through exchange of formal documents.
„ . Continuous interaction with the customer is considered to be much more important than effective contract negotiation. A customer representative is required to be a part of the development team. This facilitates close, daily co-operation between customers and developers.

Agile development projects usually deploy pair programming. Several studies indicate that programmers working in pairs produce compact well-written programs and commit fewer errors as compared to programmers working alone.

Advantages and disadvantages of agile methods

The agile methods derive much of their agility by relying on the tacit knowledge of the team members about the development project and informal communications to clarify issues, rather than spending significant amounts of time in preparing formal documents and reviewing them. Though this eliminates some overhead, but lack of adequate documentation may lead to several types of problems, which are as follows:
„ . Lack of formal documents leaves scope for confusion and important decisions taken during different phases can be misinterpreted at later points of time by different team members.
„ . In the absence of any formal documents, it becomes difficult to get important project decisions such as design decisions to be reviewed by external experts.
„ . When a project completes and the developers disperse, maintenance can become a problem

Agile *versus* Other Models

In the following subsections, we compare the characteristics of the agile model with other models of development

Agile model *versus* iterative waterfall model

The waterfall model is highly structured and makes a project to systematically step through requirements-capture, analysis, specification, design, coding, and testing stages in a planned sequence. Progress is generally measured in terms of the number of completed and reviewed artifacts such as requirements specification document, design documents, test plans, code reviews, etc. In contrast, while using an agile model, progress is measured in terms of the developed and delivered functionalities. In agile model, frequent delivery of working versions of the software is made. However, as regards to similarity it can be said that agile teams use the waterfall model on a small scale, repeating the entire waterfall cycle in every iteration.

If a project that is being developed using waterfall model is cancelled mid-way during development, then there is usually nothing to show from the abandoned project beyond several documents. With agile model however, even if a project is cancelled midway, it still leaves the customer with some worthwhile code, that might possibly have already been put into live operation.

Agile *versus* exploratory programming

Though a few similarities do exist between the agile and exploratory program development styles, there are vast differences between the two as well. Agile development model's frequent re-evaluation of plans, emphasis on face-to-face communication and relatively sparse use of documentation are similar to that of the exploratory style. Agile teams, however, do follow defined and disciplined processes and carry out systematic requirements capture and rigorous designs, as compared to the chaotic coding that takes place in exploratory programming.

Agile model *versus* RAD model

The important differences between the agile and the RAD models are the following:
„ . Agile model does not recommend developing prototypes, but emphasises systematic development of each incremental feature. In contrast, the central theme of RAD is to design quick-and-dirty prototypes, which are then refined into production quality code.
„ . Agile projects logically break down the solution into features that are incrementally developed and delivered. The RAD approach does not recommend this. Instead, developers using the RAD model focus on developing all the features of an application by first doing it badly and then successively improving the code over time.
„ . Agile teams only demonstrate completed work to the customer. In contrast, RAD teams demonstrate to customers screen mock ups, and prototypes, that usually make several simplifications such as table look-ups rather than performing actual computations.

Extreme Programming Model

Extreme programming (XP) is an important process model under the agile umbrella and was proposed by Kent Beck in 1999. The name of this model reflects the fact that it recommends taking the best practices that have worked well in the past in projects to extreme levels. This model is based on a rather simple philosophy: "If something is known to be beneficial, why not put it to constant use?" Based on this principle, it puts forward several key practices that need to be practiced to the extreme. Please note that most of the key practices that it emphasises were already recognised as good practices for quite some time.

Good practices that need to be carried on to the extreme

In the following subsections, we mention some of the good practices that have been recognized in the extreme programming model and the ways that have been suggested to maximize their use.
Code review: Code review is good since it helps to detect and correct problems most efficiently. XP suggests pair programming as the way to achieve continuous review. In pair programming, coding is carried out by pairs of programmers. In each pair, the two programmers take turn in writing code. While one writes code, the other reviews the code being written.
Testing: Testing code helps to remove bugs and improves its reliability. XP suggests Test-Driven Development (TDD) to continually write and execute test cases. In the TDD approach, test cases are written for a feature to be implemented, even before any code for it is written. That is, before starting to write code to implement a feature, test cases are written based on user stories. Writing of a piece of code is considered to be complete only after it passes these tests.
Incremental development: Incremental development is good, since it helps to get customer feedback, and extent of features delivered is a reliable indicator of progress. It suggests that the team should come up with new increments every few days.

Simplicity: Simplicity leads to good quality code. It also makes it easier to test and debug the code. Therefore, one should try to create the simplest code and make the basic functionality being written to work. For creating the simplest code, one should ignore the aspects such as efficiency, reliability, maintainability, etc. Once the simplest code works, other desirable aspects can be introduced through refactoring.
Design: Since having a good quality design is important to develop a good quality solution, every team member should perform some design on a daily basis. This can be achieved through refactoring, which involves improving a piece of working code for enhanced efficiency and maintainability.
Integration testing: Integration testing is important since it helps to identify the bugs at the interfaces of different functionalities. To this end, extreme programming suggests that the developers should achieve continuous integration. They should perform system builds as well as integration testing several times a day.

Basic idea of extreme programming model

XP is based on frequent releases (called *iterations*), during which the developers implement "user stories". User stories are similar to use cases, but are more informal and are simpler.
A user story is the conversational description by the user about a feature of the required system. For example, a set of user stories about a library software can be: A library member can issue a book.
A library member can query about the availability of a book.
A library member should be able to return a borrowed book.
On the basis of user stories, the project team proposes "metaphors"—a common vision of how the system would work. This essentially is the architectural design (also known as high-level design). The development team may decide to construct a spike for some feature. A spike, is a very simple program that is constructed to explore the suitability of a solution being proposed. A spike can be considered to be similar to a prototype.
XP prescribes several basic activities to be part of the software development process. We discuss these activities in the following subsections:
Coding: XP argues that code is the crucial part of any system development process, since without code it is not possible to have a working system. Therefore, utmost care and attention need to be placed on coding activity. However, the concept of code as used in XP has a slightly different meaning from what is traditionally understood. For example, coding activity includes drawing diagrams (modelling) that will be transformed to code, scripting a web-based system, and choosing among several alternative solutions.
Testing: XP places high importance on testing and considers it be the primary means for developing a fault-free software.
Listening: The developers need to carefully listen to the customers if they have to develop a good quality software. Programmers may not necessarily have an in-depth knowledge of the specific domain of the system under development. On the other hand, customers usually have this domain knowledge. Therefore, for the programmers to properly understand the required functionalities of the system, they have to listen to the customer.
Designing: Without a proper design, a system implementation becomes too complex and the dependencies within the system become too numerous and it becomes very difficult to comprehend the solution. This makes any maintenance activity prohibitively expensive.
A good design should result in elimination of complex dependencies within the system.
With this intention, effective use of a suitable design technique is emphasised.
Feedback: It espouses the wisdom: "A system staying isolated from the scrutiny of the users is trouble waiting to happen". It recognizes the importance of user feedback in understanding the exact customer requirements. The time that elapses between the development of a version and collection of feedback on it is critical to learning and making changes. It argues that frequent contact with the customer makes the development effective.

Simplicity: A corner-stone of XP is the principle: "build something simple that will work today, rather than trying to build something that would take time and yet may never be used". This in essence means that attention should be than devoting time and energy on speculations about future requirements.

Applicability of extreme programming model

The following are some of the project characteristics that indicate the suitability of a project for development using extreme programming model:

Projects involving new technology or research projects: In this case, the requirements change rapidly and unforeseen technical problems are often needed to be resolved.

Small projects: Extreme programming was proposed in the context of small teams, as face-to-face communication is easier to achieve in this situation.

Project characteristics not suited to development using agile models

The following are some of the project characteristics that indicate unsuitability of agile development model for use in a development project:

Stable requirements: Conventional development models are more suited to use in projects characterised by stable requirements. For such projects, it is known that almost no changes to the gathered requirements will occur. Therefore, process models such as iterative waterfall model that involve making long-term plans during project initiation can meaningfully be used.

Mission critical or safety critical systems: In the development of such systems, the traditional SDLC models are usually preferred to ensure the required reliability.

Scrum

Scrum is one of the agile development models. In the scrum model, the entire project work is divided into small work parts that can incrementally be developed and delivered over time boxes. These time boxes are called *sprints*. At the end of each sprint, the stakeholders and the team members meet to assess the developed software increment. The stakeholders may suggest any changes an improvement to the developed software that they might feel necessary in scrum, the software gets developed over a series sprints. In each sprint, manageable increments to the software (or chunks) are deployed at th client site and the client feedback is obtained after each sprint.

Key roles and responsibilities

In the scrum model, the team members assume three basic roles: product owner, scrum master, and team member. The responsibilities associated with these three basic roles are discussed as follows:

Product owner: The product owner represents the customer's perspective and guides the team toward building the right software. In other words, the product owner is responsible for communicating the customer's perspective of the software to the development team. To this end, in every sprint, the product owner in consultation with the team members defines the features of the software to be developed in the next sprint, decides on the release dates, and also may reprioritize the required features that are yet to be developed if necessary.

Scrum master: The scrum master acts as the project manager for the project. The responsibilities of the scrum master include removing any impediments that the project may face, and ensuring that the team is fully productive by fostering close cooperation among the team members. In addition, the scrum master acts as a liaison between the customers, top management, and the team and facilitates the development work. The scrum team is, therefore, shielded by the scrum master from external interferences.

Team member: A scrum team usually consists of cross-functional team members with expertise in areas such as quality assurance, programming, user interface design, and testing. The team is self-organising in the sense that the team members distribute the responsibilities among themselves. This is in contrast to a conventional team where a designated team leader decides who will do what.

Artifacts

Three main artifacts form an important part of the scrum methodology. These are: product backlog, sprint backlog, and sprint burndown chart. We briefly describe the purpose and contents of these three documents in the following subsections.

Product backlog: This document at any time during development contains a prioritized listing of all the features that are yet to be developed. In this document, the features of the software are usually written in the form of user stories. During the course of development, new features may get added to the product backlog and some features may even get deleted. This document is, therefore, a dynamic document that is periodically updated and reprioritised. The product backlog forms the pool of features from which some of the features are chosen for development during the next sprint.

Sprint backlog: Before every sprint, a sprint planning meeting takes place. During a sprint planning meeting, the team identifies one or more features (user stories) from the product backlog and decides on the tasks that would be needed to be undertaken for the development of the identified features and this forms the sprint backlog. In other words, we can say that the sprint backlog lists the tasks that are identified and committed by the team to develop and complete during the ensuing sprint.

Sprint burndown chart: During a sprint, the sprint burndown chart is used as a tool to visualize the progress made and the work remaining to be undertaken on a daily basis. At the end of a team meeting called the daily stand-up meeting, the scrum master determines the work completed in the previous day and the list of work that remains to be completed and represents these in the sprint burndown chart. The sprint burndown chart gives a visual representation of the progress achieved and the work remaining to be undertaken. The horizontal axis of the sprint burndown chart represents the days in the sprint, and the vertical axis shows the amount of work remaining (in hours of effort) at the end of each day. An example sprint burndown chart has been shown in Figure 2.10. As can be seen, the chart visually represents the progress made on a day-to-day basis and the work that is remaining to be completed. This chart, therefore, helps to quickly determine whether the sprint is on schedule and whether all the planned work would fi nish by the desired date.

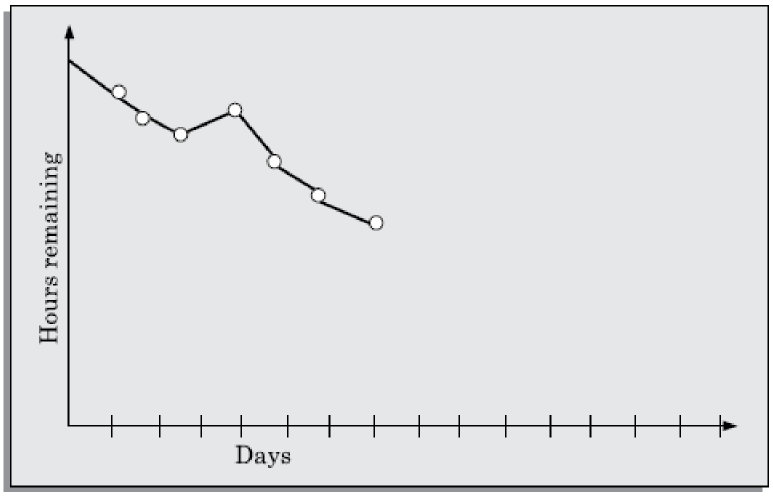


FIGURE 2.10 An example of a sprint burndown chart.

Scrum ceremonies

The term *scrum ceremonies* is used to denote the meetings that are mandatorily held during the duration of a project. The scrum ceremonies include three different types of meetings sprint planning, daily scrum, and sprint review meeting.

Sprint planning: During the sprint planning meeting, the team members commit to develop and deliver certain features in the ensuing sprint, out of those listed I the product backlog. In this meeting, the product owner works with the team to negotiate which product backlog items the team should work on in the ensuing sprint in order to meet the release goals. It is the responsibility of the scrum master to ensure that the team agrees to realistic goals for a sprint.

Daily scrum: The daily scrum is a short stand-up meeting conducted during ever day morning to review the status of the progress achieved so far and the major issue being faced on a day-to-day basis. The daily scrum meeting is not a problem-solving meeting, rather each member updates the teammates about what he/she has achieved in the previous day. Each team member focuses on answering three questions: What did he/she do yesterday? What will he/she do today? What obstacles is he/she facing? The daily scrum meeting helps the scrum master to track the progress made so far and helps to address any problems needing immediate attention.

Sprint review meeting: At the end of each sprint, a sprint review is conducted. In this meeting, the team demonstrates the new functionality developed during the sprint that was just completed to the product owner and to the stakeholders. Feedback is collected from the participants of the meeting and these either are taken into account in the next sprint or are added to the product backlog.

Lean Software Development

The origin of the Lean software development process can be traced to the Lean process that was developed in a car manufacturing industry (Toyota production system). It was soon found to be an effective product development process that holds significant benefits. It soon became widely accepted and found use in many other types of product manufacturing industries. The central theme of Lean is to achieve overall process efficiency through elimination of various things that cause waste of work and introduce delays. Kanban methodology was developed as a part of the Lean approach to visualize the work flow. Using Kanban, it becomes easy to identify and eliminate potential delays and bottlenecks.

Due to its obvious advantages, the Kanban methodology has now become widely accepted and has emerged as a popular methodology on its own standing and is being used in conjunction with other agile methodologies such as XP.

Lean methodology in software development

About a decade back, Lean methodology was adopted for use in the software development projects and has since gained significant popularity. In software development projects, it has been found effective in reducing delays and bottlenecks by minimizing waste. Please note that in the Lean methodology, any work that does not add value to the customer is considered to be waste. A few examples of waste are rework required for defect correction, gold plating and scope creep, delay in staffing, and excessive documentation.

Kanban

Kanban is a Japanese word meaning a sign board. The main objective of Kanban is to provide visibility to the workflow. This in turn is expected to help debottleneck congestions in the process and minimize delays. Though developed in the context of general product manufacturing industries, Kanban has been observed to be highly appropriate in software development projects. A possible reason behind this could be that in manufacturing, the process is visible as the work product progresses along the production line. But, in software development, the visibility of the product, the product line, and the work performed are low. In this context, the Kanban method helps a project team to visualise the workflow, limit work in progress (WIP) at each workflow stage, and measure cycle time.

Kanban board and Kanban card

A principal artifact of the Kanban approach is the Kanban board. A Kanban board is typically a white board on which sticky notes called Kanban cards are attached. Kanban cards represent the work items that are waiting for completion at different development stages. A card is moved on the board as the corresponding work is completed at a stage. This provides visualization of the work fl ow and the work queued up at different stages. The process steps are called stages or work stations, and are drawn as columns on the Kanban board. On a Kanban board, the work process causing delay is easy to spot, and remedial actions can be initiated. An example Kanban board has been shown in Figure 2.11.

One of the main premises of the Kanban approach is that during development when some work stage becomes overloaded with work, it becomes a bottleneck and introduces delays; thereby slowing down the entire development process. This eventually shows up as schedule slips. Kanban board helps to identify the bottlenecks in a development process early, so that corrective actions can be taken in time. In the Kanban board, the different stages of development such as coding, code review, and testing are represented by different columns in the board. The work itself is broken down into small work items and the work item name is written on a card. The card is stuck to the appropriate column of the Kanban board to represent the stage at which the work item is waiting for completion. As the work item progresses through different stages, the card is moved accordingly on the Kanban board. In Kanban, it is a requirement that the number of work items (represented as cards) that can be in progress at any point of time is limited. We can, therefore, say that Kanban enforces work in progress limits (WIP). The general idea behind it is to reduce the workload at a processing station, so that the developers can concentrate only on a few tasks and complete those before taking up anything new. The goal of limiting WIP is to reduce stress and to raise productivity as well as product quality.

Visibility provided by Kanban board

The Kanban board provides visibility to the software process. It shows work assigned to each developer at any time and the work flow. It, therefore, becomes possible to define and communicate the priorities and the bottlenecks get highlighted. The average time it takes to complete a work item (sometimes called the *cycle time*) is tracked and optimized so that the process becomes efficient and predictable. The Kanban system, therefore, helps to limit the amount of work in process so that the work flowing through the system matches its full capacity.

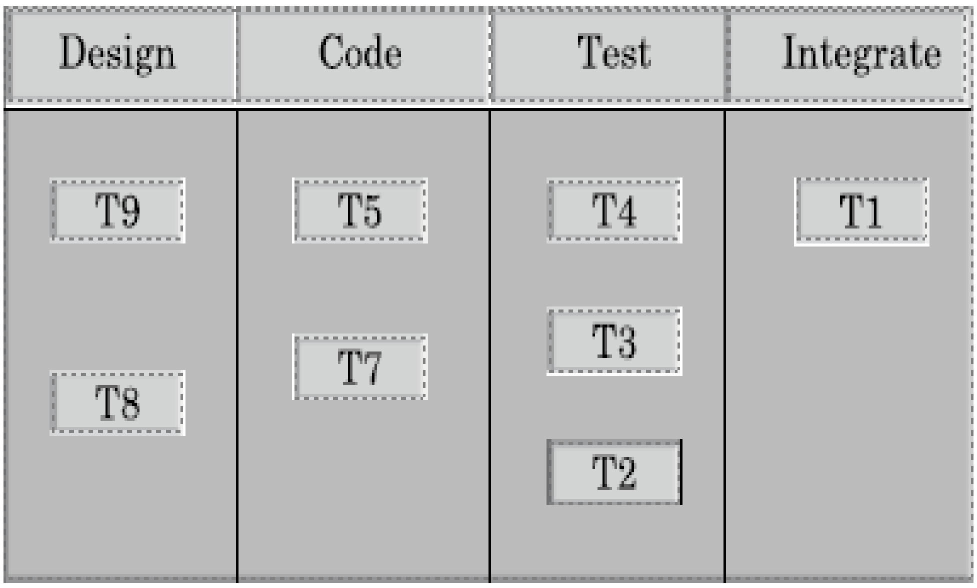


FIGURE 2.11 An example Kanban board.

SPIRAL MODEL

This model gets its name from the appearance of its diagrammatic representation that looks like a spiral with many loops (see Figure 2.12). The exact number of loops of the spiral is not fixed and can vary from project to project. The number of loops shown in Figure 2.12 is just an example. Each loop of the spiral is called a *phase* of the software process. The exact number of phases through which the product is developed can be varied by the project manager depending upon the project risks. A prominent feature of the spiral model is handling unforeseen risks that can show up much after the project has started. In this context, please recollect that the prototyping model can be used effectively only when the risks in a project can be identified upfront before the development work starts. As we shall discuss, this model achieves this by incorporating much more flexibility compared to SDLC of other models.

Risk handling in spiral model

A risk is essentially any adverse circumstance that might hamper the successful completion of a software project. As an example, consider a project for which a risk can be that data access from a remote database might be too slow to be acceptable by the customer. This risk can be resolved by building a prototype of the data access subsystem and experimenting with the exact access rate. If the data access rate is too slow, possibly a caching scheme can be implemented or a faster communication scheme can be deployed to overcome the slow data access rate. Such risk resolutions are easier done by using a prototype as the pros and cons of an alternate solution scheme can evaluated faster and inexpensively, as compared to experimenting using the actual software application being developed. The spiral model supports coping up with risks by providing the scope to build a prototype at every phase of software development.

Phases of the Spiral Model

Each phase in this model is split into four sectors (or quadrants) as shown in Figure 2.12. In the first quadrant, a few features of the software are identified to be taken up for immediate development based on how crucial it is to the overall software development. With each iteration around the spiral (beginning at the center and moving outwards), progressively more complete versions of the software get built. In other words, implementation of the identified features forms a phase.

Quadrant 1: The objectives are investigated, elaborated, and analysed. Based on this, the risks involved in meeting the phase objectives are identified. In this quadrant, alternative solutions possible for the phase under consideration are proposed.

Quadrant 2: During the second quadrant, the alternative solutions are evaluated to select the best possible solution. To be able to do this, the solutions are evaluated by developing an appropriate prototype.

Quadrant 3: Activities during the third quadrant consist of developing and verifying the next level of the software. At the end of the third quadrant, the identified features have been implemented and the next version of the software is available.

Quadrant 4: Activities during the fourth quadrant concern reviewing the results of the stages traversed so far (i.e. the developed version of the software) with the customer and planning the next iteration of the spiral. In the spiral model of development, the project manager dynamically determines the number of phases as the project progresses. Therefore, in this model, the project manager plays the crucial role of tuning the model to specific projects. To make the model more efficient, the different features of the software that can be developed simultaneously through parallel cycles are identified. To keep our discussion simple, we shall not delve into parallel cycles in the spiral model.

Advantages/pros and disadvantages/cons of the spiral model

There are a few disadvantages of the spiral model that restrict its use to only a few types of projects. To the developers of a project, the spiral model usually appears as a complex model to follow, since it is risk-driven and is more complicated phase structure than the other models we discussed. It would therefore be counterproductive to use, unless there are knowledgeable and experienced staff in the project. Also, it is not very suitable for use in the development of outsourced projects, since the software risks need to be continually assessed as it is developed. In spite of the disadvantages of the spiral model that we pointed out, for certain categories of projects, the advantages of the spiral model can outweigh its disadvantages.

In this regard, it is much more powerful than the prototyping model. Prototyping model can meaningfully be used when all the risks associated with a project are known beforehand. All these risks are resolved by building a prototype before the actual software development starts.

Spiral model as a meta model

As compared to the previously discussed models, the spiral model can be viewed as a *meta model*, since it subsumes all the discussed models. For example, a single loop spiral actually represents the waterfall model. The spiral model uses the approach of the prototyping model by first building a prototype in each phase before the actual development starts. These prototypes are used as a risk reduction mechanism. The spiral model incorporates the systematic step-wise approach of the waterfall model. Also, the spiral model can be considered as supporting the evolutionary model—the iterations along the spiral can be considered as evolutionary levels through which the complete system is built. This enables the developer to understand and resolve the risks at each evolutionary level (i.e. iteration along the spiral).