Table of Contents

Chapter 1

Software Development and Object Oriented Programming Paradigms

This chapter presents various methodologies for problem solving and development of applications that have evolved over a period of time. This is primarily driven by the increasing complexity of software and the cost of software maintenance growing rapidly. The chapter introduces object-oriented design and programming as a silver bullet to solve software crisis. It then discusses various features of objected oriented programming (OOP) from encapsulation and inheritance to templates. Finally, the chapter presents various OOP programming languages with their unique properties.

Objectives

After learning the contents of this chapter, the reader must be able to:

- understand programming paradigms
- know the factors influencing the complexity of software development
- define software crisis
- know the important models used in software engineering
- explain the natural way of solving a problem
- understand the concepts of object-oriented programming
- define abstraction and encapsulation
- differentiate between interface and implementation
- understand classes and objects
- state the design strategies embedded in OOP
- compare structured programming with OOP
- list examples of OOP languages
- list the advantages and applications of OOP

1.1 Introduction

Computers are used for solving problems quickly and accurately irrespective of the magnitude of the input. To solve a problem, a sequence of instructions is communicated to the computer. To communicate these instructions, *programming* languages are developed. The instructions written in a programming language comprise a *program*. A group of programs developed for certain specific purposes are referred to as *software* whereas the electronic components of a computer are referred to as *hardware*. Software activates the hardware of a computer to carry out the desired task. In a computer, hardware without software is similar to a body without soul. Software can be system software or application software. *System software* is a collection of system programs. A *system program* is a program, which is designed to operate, control and utilize the processing capabilities of the computer itself effectively. *System programming* is the activity of designing and implementing system programs. Almost all the operating systems come with a set of ready to use system programs: user management, file system management, and memory management. By composing programs it is possible to develop new, more complex, system programs. *Application software* is a collection of prewritten programs meant for specific applications.

Computer hardware can understand instructions only in the form of machine codes i.e. 0's and 1's. A programming language used to communicate with the hardware of a computer is known as *low-level language* or *machine language*. It is very difficult for humans to understand machine language programs because the instructions contain a sequence of 0’s and 1’s only. Also, it is difficult to identify errors in machine language programs. Moreover, low-level languages are machine dependent. To overcome the difficulties of machine languages, *high-level languages* such as Basic, Fortran, Pascal, COBOL and C were developed.

High-level languages allow some English-like words and mathematical expressions that facilitate better understanding of the logic involved in a program. While solving problems using high-level languages, importance was given to develop an *algorithm* (step by step instructions to solve a problem). While solving complex problems, a lot of difficulties were faced in the algorithmic approach. Hence *object-oriented programming languages* such as C++ and Java were evolved with a different approach to solve the problems. Object-oriented languages are also high-level languages with concepts of classes and objects that are discussed later in this chapter.

1.2 Problem Domain and Solution Domain

A *problem* is a functional specification of desired activities to generate the intended output. A *solution* is the method of achieving the desired output. For example, *getting a train-ticket from Chennai to Delhi* is a problem statement and *purchasing a ticket by going to the Reservation Ticket Counter* is a solution to the problem. The output of this problem is the reserved ticket. Every problem belongs to a domain of knowledge. The domain is the general field of business or technology in which the user will use the software. The domain knowledge for reserving the ticket requires knowing the train routes and fares to do that task. Hence, the term *problem domain* is used in problem solving. The domain or the sector to which the problem belongs defines problem domain. The problem that specifies the requirement in a particular knowledge domain and the domain experts associated with the task of explaining the requirements belong to problem domain. Similarly the solution obtained belongs to the *solution domain*. The subject matter that is of concern to the computer and the persons associated with the task of devising solution define solution domain. The problem domain specifies the scope of the problem along with the functional requirements represented in a high level so that human beings can understand.

The solution domain contains the procedures or techniques used to generate the desired output by a computer. Thus, *problem solving* is a mapping of problem domain to solution domain as shown in Figure 1.1. It is the act of finding solution to a problem. The formulation of solution for a simple problem is easy. The solution for simple problems may not require any systematic approach. But a complex problem requires logical thinking and careful planning. Generally the problems to be solved using computers will be reasonably complex.

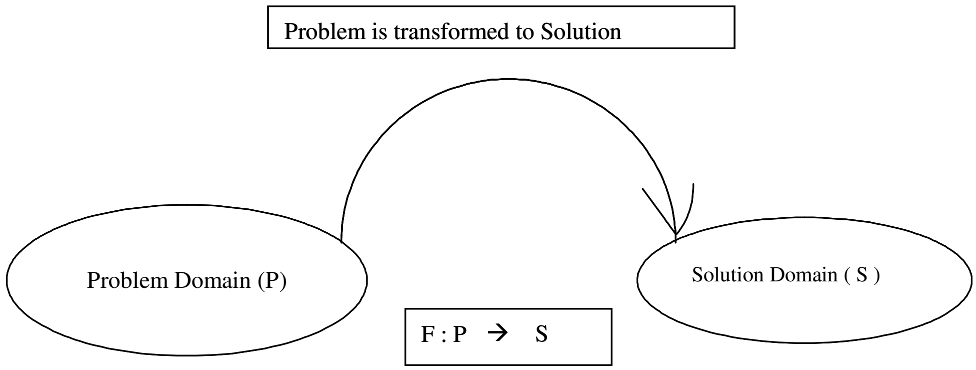

Figure 1.1: Problem Solving

1.2.1 Problem States

The problem has a start state and an end state or goal state. The solution helps the transition from the start state to the end state as shown in Figure 1.2. It defines the sequence of actions that produces the end state from the start state.

Figure 1.2: Solution to a problem

The states are to be clearly understood before trying to get a solution for the problem. The initial conditions and assumptions are to be explicitly stated to derive a solution for a problem. The solution to a problem must be viewed in terms of people associated with it.

1.3 Types of Persons Associated to Solution

We may observe the three types of people associated with a solution to a problem as shown in Figure 1.3. The logical solution may be explained by the domain experts. A domain expert is a person who has a deep knowledge of the domain. The program is developed by one set of people and the same is used by another set of people. The people developing solution are called *developers* and the people using the solution are called *users*. The developer is also known as *supplier* or *programmer* or *implementer*. The user is also called *client* or *customer* or *end-user*. The solution represents the instructions to be followed to generate the output. The solution of a problem should be carefully planned to enable the user to gain confidence in the solution.

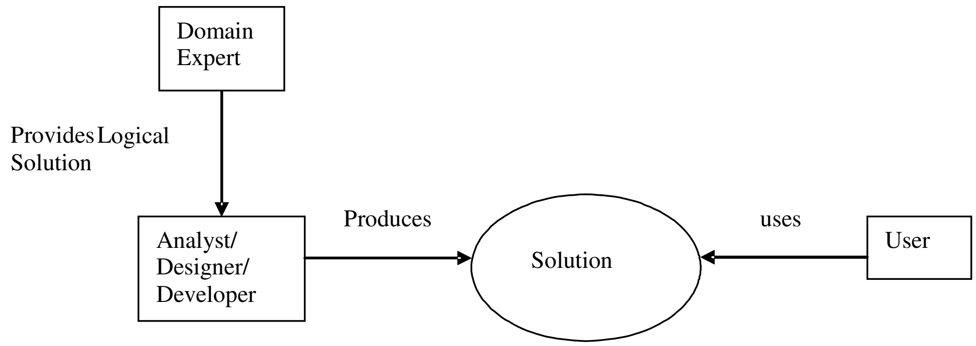

Figure 1.3: People associated with solution

1.4 Program

The solution to a problem is written in the form of a *program,* while a computer is used to solve the problem. A program is a set of instructions written in a *programming language*. A programming language provides the medium for conveying the instructions to the computer. There are many programming languages such as BASIC, FORTRAN, Pascal, C, C++, etc., similar to the written languages like English, Tamil and Hindi. Once the steps to be followed for solving a problem are identified, it is easier to convert these steps to a program through a programming language. The idea of providing solution is quite challenging. The domain experts play a major role in formulating the solution. The formulation of solution is important before writing a program. It requires logical thinking, careful planning and systematic approach. This can be achieved through the proper combination of domain experts, system analysts/system designers and developers. The program takes the input from the user and generates the desired output as shown in Figure 1.4.

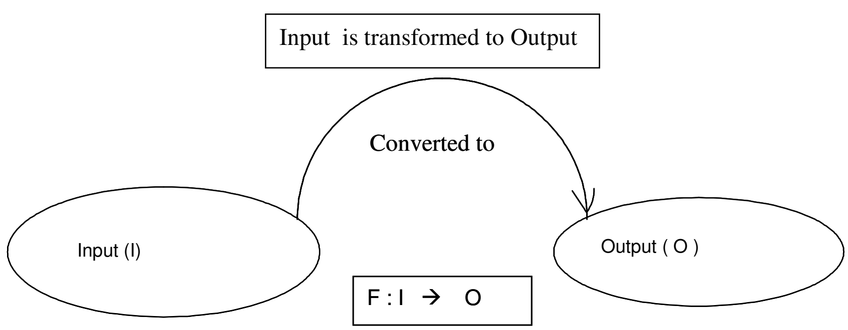

Figure 1.4: Program

1.5 Approaches in Problem Solving

The principles and techniques used to solve a problem are classified under the following categories. The following strategies are used in building solutions to a problem.

1.5.1 Multiple attacks or Ask Questions

By asking questions like what, why and how, the solution may be outlined for some problems. Questions can be asked to many people irrespective of the domain and the answers to multiple attacks of questions may help in revealing the solution. Whenever the solution is not known, this approach may be used.

1.5.2 Look for things that are similar

We should never reinvent the wheel again. The existing solution for a similar problem can be used to solve a problem. For example, finding the maximum value in a set of numbers is the same as finding the maximum mark in a class of students or finding the highest temperature in a day. All these different problems require the same concept of finding the biggest value among the values. The solution is based on the similar nature of a problem.

1.5.3 Working backward or bottom-up approach

The problem can also be solved by starting from the *Goal* state and reaching the *Start* state. For example, sometimes we prefer to derive an equation in mathematics from right side to left side. The solution is derived in the reverse direction. For complex problems, this approach will be an easier approach. Consider the problem of reaching an unknown place from a known place. It is always easier to trace a known place starting from an unknown place compared to tracing from known to unknown place. There may be many known landmarks nearer to the known place helping in locating the place. If any one such landmark is reached, it is equivalent to finding the solution. But, the landmarks of the unknown place are new while searching. Hence, even by reaching to the nearest place, sometimes the location may not be identified and the tracing becomes difficult.

1.5.4 Problem decomposition or top-down approach

The problem is decomposed into small units and they are further decomposed into smaller units over and over again until each smaller unit is manageable. The complex problem is simplified by decomposing it into many simple problems. It is applicable for simple and fairly complex problems. The top-down approach is also known as stepwise refinement or modular decomposition or structured approach or algorithmic approach.

1.6 Styles of Programming

Each programming language enforces a particular style of programming. The way of organizing information is influenced by its style of programming and it is known as *programming paradigm*. *First generation programming languages* (1954-1958) such as FORTRAN I, ALGOL 58 and FLOWMATIC were used for numeric computations. Any program makes use of data. Data is represented by a variable or constant in a program. To perform an action, an operator acts on the data (operand). Operands and operators are combined to form expressions. Each instruction is written as a statement with the help of expressions. A sequence of statements comprises a program. The structure of first generation languages is shown in Figure 1.5.

There is no support for subprograms. Such programming is known as *monolithic programming*. The data is globally available and hence there is no chance of *data hiding* (denying the access of data is known as data hiding). First generation languages were used only for simple applications. The program is closer to solution domain by representing the operations/operators in the programming language that can be performed in the computer.

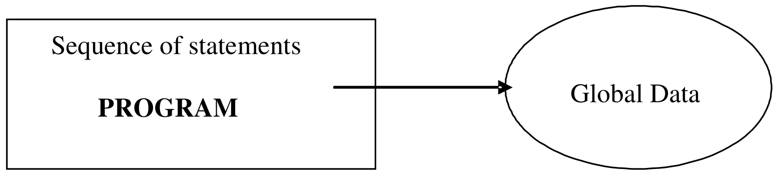

Figure 1.5: Structure of the first generation languages

Second generation programming languages (1959-1961) introduced subprograms (functions or procedures or subroutines) as shown in Figure 1.6. Inclusion of subprograms avoids repetition of coding. Such programming is known as *procedural programming*. Second generation language is suitable for applications that require medium sized programs.

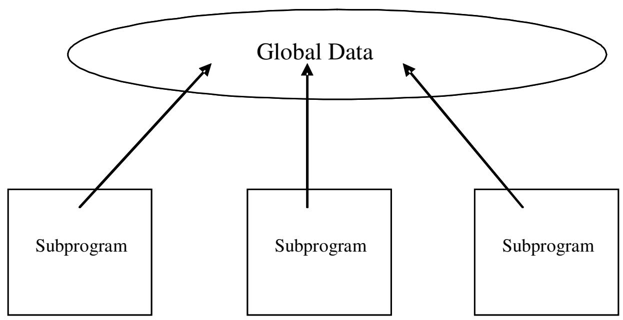

Figure 1.6: Structure of the second generation languages

FORTRAN II, ALGOL 60 and COBOL are second generation languages. The second generation languages provided the possibility of *information hiding* (i.e., hiding the implementation details of a subprogram). However, sharing the same data by many subprograms breaks the data hiding principle. Hence, data hiding is only partially succeeded. Here also the program is closer to solution domain where concentration is on operations/operators using functions.

Third generation programming languages (1962-1970) such as PASCAL and C use sequential code, global data, local data and subprograms as shown in Figure 1.7. They follow *structured programming*, which supports modular programming. The program is divided into a number of *modules*. Each module consists of a number of subprograms represented by rectangles.

Importance was given for developing an algorithm and hence this approach is also known as *algorithmic oriented programming*. In structural programming approach, data and subprograms exist separately (Algorithms + Data Structures = Programs). A main program calls the subprograms. *Structured programming* approach supports the following features:

1. Each procedure has its own local data and algorithm.
2. Each procedure is independent of other procedures.
3. Parameter passing mechanisms are evolved.
4. It is possible to create user defined data types.
5. A rich set of control structures is introduced.

6. Scope and visibility of data are introduced.
7. Nesting of subprograms is supported.
8. Procedural abstractions or function abstractions are achieved yielding abstract operations.
9. Subprograms are the basic physical building blocks supporting modular programming.

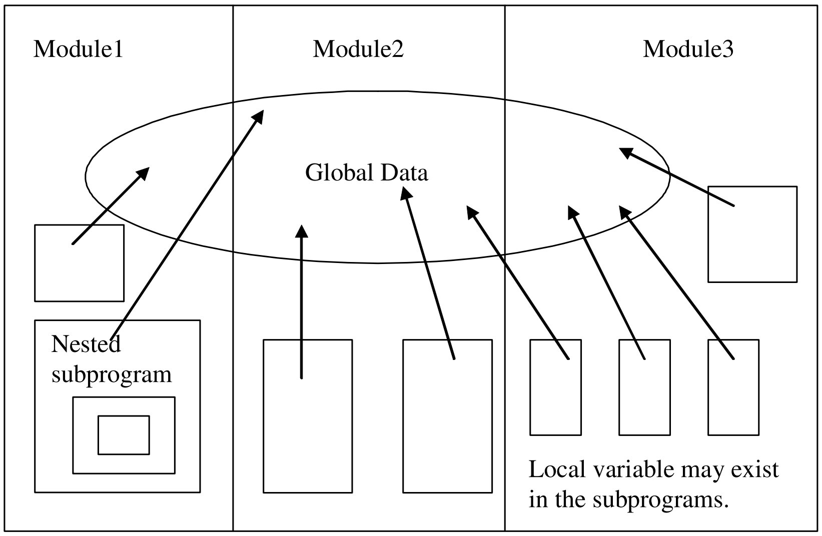

Figure 1.7: Data in third generation programming languages

By introducing *scopes*[1] of variables, data hiding was made possible. For a very complex problem, the maintenance of the program becomes very tedious because of the existence of so many subprograms and global data. Here also the program is closer to the solution domain.

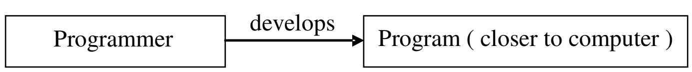

Figure 1.8: Relationship between a program and programmer

It can be observed that in structured programming, the emphasis is on the subprograms and the efficient way of developing *algorithms* in terms of computing time and computer memory to solve the problem. The relationship between programmer and program is given prime importance as shown in Figure 1.8. *Hence structured programming paradigms depend on solution domain and not on problem domain*. The data is not given importance regarding access permission.

To solve a complex problem using *top-down approach*, first the complex problem is decomposed into smaller problems. Further these smaller problems are decomposed and finally a collection of small problems are left out. Each problem is solved one at a time. Structured programming starts with high-level descriptions of the problem representing global functionality. It successively refines the global functionality by decomposing it into subprograms using lower level

[1] A scope identifies the portion of source program from which a variable can be accessed. It normally consists in the portion of text that starts from the variable declaration and spans till the end of the nearest enclosing block.

descriptions, always maintaining correctness at each level. At each step, either a control or a data structure is refined. Thus *top-down* approach is followed in structured programming. This is a fairly successful approach because it will cause problems only when there is a revision of design phase. Such revisions may result in massive changes in the program. Also the possibility of *reuse of software modules* is minimized**.**

There was a generation gap from 1970 to 1980. Many programming languages evolved, but only a few of them were used in software development. Despite the invention of new programming languages and software engineering concepts, software industries were unable to meet the demand in reality.

1.7 Complexity of Software

Mainly simple problems were solved using computers during the initial evolution phases of computing technologies (prior to 1990). These days, computers are utilized in solving many mission critical problems and they are playing a vital role in the fields of space, defense, research, engineering, medicine, industry, business and even in music and painting. For example, Inter-Continental Ballistic Missiles (ICBM) in defense and launching of satellites in space cannot be controlled without computers. Such applications cannot be even imagined without computers. Influence of computers in various activities leads to the establishment of many software companies engaged in the development of various types of applications.

Large projects involve many highly qualified persons in the software development process. Software industries face a lot of problems in the process of software development. The following factors influence the complexity of software development as shown in Figure 1.9.

1. Improper understanding of the problem

The users of a software system express their needs to the software professionals. The requirement specification is not precisely conveyed by the users in a form understandable by the software professionals. This is known as impedance mismatch between the users and software professionals.

2. Change of rules during development

During the software development process because of some government policy or any other industrial constraints realized, the users may request the developer to change certain rules of the problem already stated.

3. Preservation of existing software

In reality, the existing software is modified or extended to suit the current requirement. If a system had been partially automated, the remaining automation process is done by considering the existing one. It is expensive to preserve the existing software because of the non-availability of experts in that field all the time. Also it results in complexity while integrating newly developed software with the existing one.

4. Management of development process

Since the size of the software becomes larger and larger in the course of time it is difficult to manage, coordinate, and integrate the modules of the software.

5. Flexibility due to lack of standards

There is no single approach to develop software for solving a problem. Only standards can bring out uniformity. Since only a few standards exist in the software industries, software development is a laborious task resulting in complexity.

6. Behavior of discrete systems

The behavior of a continuous system can be predicted by using the existing laws and theorems. For example, the landing of a satellite can be predicted exactly using some theory even though it is a complex system. But, computers have systems with discrete states during execution of the software. The behavior of the software may not be predicted exactly because of its discrete nature. Even though the software is divided into smaller parts, the phase transition cannot be modeled to predict the output. Sometimes an external event may corrupt the whole system. Such events make the software extremely complex.

7. Software testing

The number of variables, control structures and functions used in the software are enormous. The discrete nature of the software execution modifies a variable and it may be unnoticed. This may result in unpredictable output. Hence, vigorous testing is essential. It is impossible to test each and every aspect of the software in a complex software system. So only important aspects are subjected to testing and the user must be satisfied with this. The reliability of the software depends on rigorous testing. But testing processes make software development more and more complex.

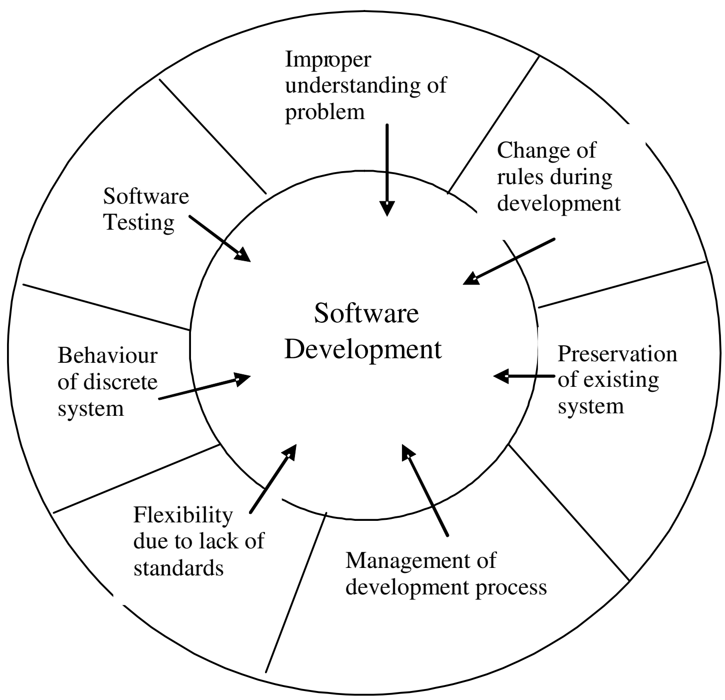

Figure 1.9: Factors influencing software complexity

1.8 Software Crisis

The complexity involved in the software development process led to the *software crisis*. Late completion, exceeding the budget, low quality, software not satisfying the stated demand and lack of reliability are the symptoms of software crisis. *Software crisis* has been the result of a missing methodology in software development. The lack of structured and organized approach to software development – not conceived as a process – led to late completion, exceeding budget in the case of large and complex project. The OO paradigm arose as a consequence of a *software crisis*, where the relative cost of software has increased substantially at a rate where software maintenance and software development cost has far outstripped that of hardware costs. This rate of increase is depicted in Figure 1.10. *Software crisis* as a term arose from the understanding that costs in software development and maintenance have increased significantly, and that software engineering concepts and innovations have not resulted in significant improvements in the productivity of software development and maintenance. The software crisis provided an impetus to develop principles and tools in software to drive, maintain and provide solid paradigms to apply to the software development life cycle, with the intent to create more reliable and reusable systems. The sharp increase in software maintenance from 1995-2000 is attributed to Y2K (Year 2000) problem in software applications. As a result Indian software engineers have gained world-wide popularity, which has in turn led to rapid growth of IT industries in India.

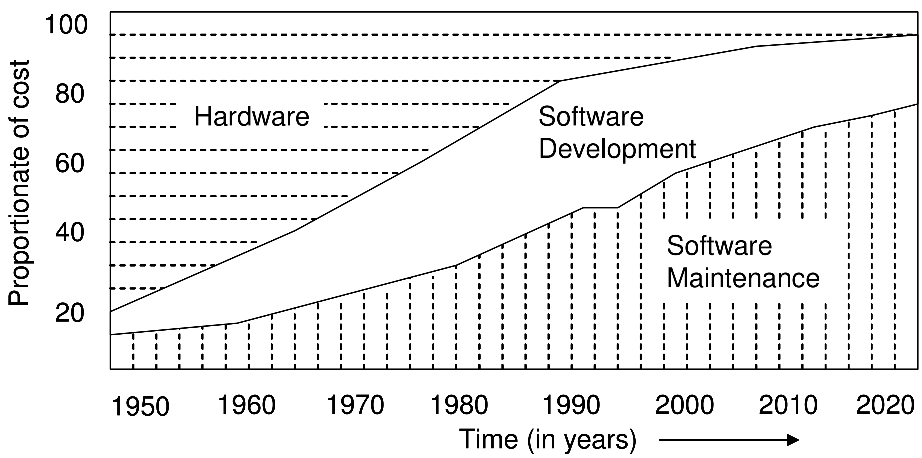

Figure 1.10: System development cost

Hardware development has been tremendously larger compared to software development. Hardware industries develop their products by assembling standardized hardware components such as integrated silicon chips. If a component fails, it is replaced by a new component without affecting the functionality of the product. Standardized components are reused in developing other products also. This revolutionary approach of *reusable components* and *easier maintenance* influenced the software development process.

1.9 Software Engineering Principles

To avoid the software crisis, software engineering principles, programming paradigms and suitable supporting software tools are introduced. Software engineering principles help to develop software in a scientific manner. Systematic engineering principles and techniques such as model building, simulation, estimation, and measurement are used to build software products. There are six main software engineering activities in the *Software Development Life Cycle* (SDLC) as shown in Figure 1.11. This model is known as *Waterfall model*.

Waterfall model follows the activities in a rigid sequential manner. There is no overlap of activities in this model. Each activity is followed after completion of the previous activity. Because of the rigid sequential nature there is a lack of iterations of activities. The analyst may use dataflow diagrams (DFDs), the designer may focus on hierarchy charts, and the programmer may use flowcharts and hence there are disjoint mappings among the SDLC activities. Generally, the analyst uses top-down functional decomposition while solving a problem. *The programmer implements the solution easily by using the procedural languages/structural programming languages that support functional decomposition*. The difficulty of *reuse* of software components still persists.

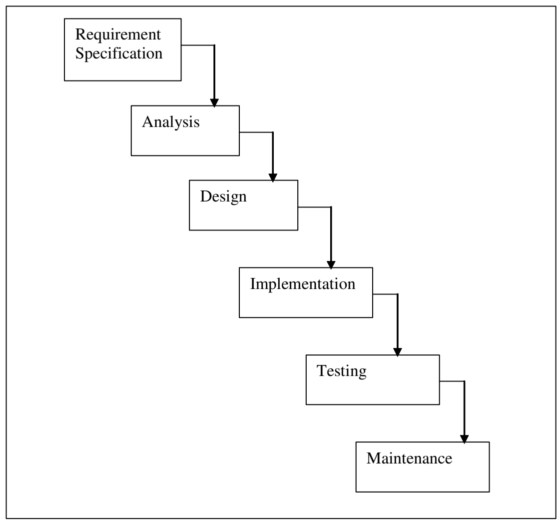

Figure 1.11: Software development activities (Waterfall Model)

Percentage of costs incurred during the different phases of SDLC is shown in Figure 1.12. Cost factor of the first two phases can be combined. It can be observed that the maintenance of software is 60% whereas all the other costs are only 40%. Hence, *maintenance* is an important factor to be considered in software development process. Also, earlier programming languages did not support reusability. An existing program cannot be reused because of the dependence of the program on its environment. Thus, the following two major problems demanded a new programming approach:

1. *Software maintenance.*
2. *Software reuse.*

Logical improvement to the *Waterfall model* resulted in the *Fountain model*. The same six activities in the software development are still followed in the same sequence. However, there is an *overlap of activities* and *iteration of activities* as shown in Figure 1.13. The *Fountain model* is a graphical representation to remind us that although some life cycle activities cannot start before others, there is a considerable overlap and merging of activities across the full life cycle. In a fountain, water rises up the middle and falls back, either to the *pool* below or is re-entrained at an intermediate level.

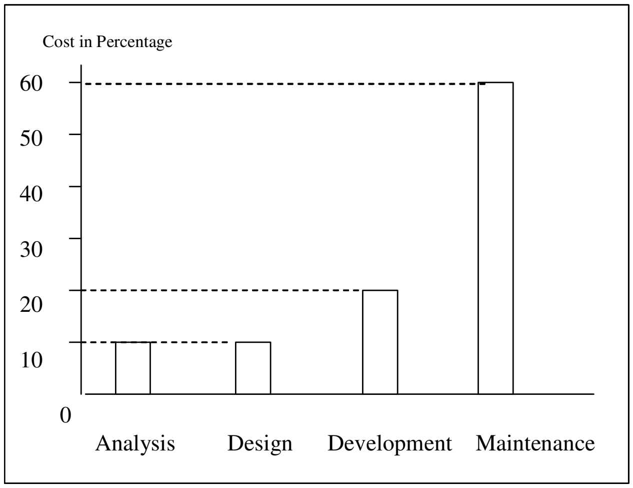

Figure 1.12: Costs involved in SDLC

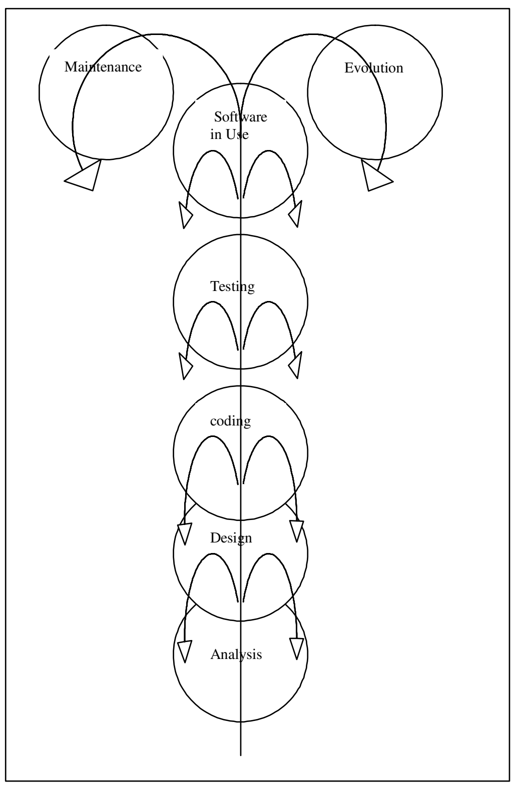

Figure 1.13: Fountain model

The Fountain model outlines the general characteristics of the systems level perception of an object-oriented development. There is a high degree of merging in the analysis, design, implementation and unit testing phases. Moving through a number of steps, falling back one or more steps and performing repeatedly, is a far more flexible approach than the one proposed by Waterfall model. It follows a *bottom up* approach, which starts from the solution. If there is an existing solution, that solution is studied first and the necessary details are identified and organized in a suitable manner. For a problem not having a solution, the domain experts (i.e., experts who are capable of providing useful information and future requirements) are consulted with the conventional solution to start with. Since the software is developed by analyzing the solution first, this approach is known as bottom up approach. There is another approach similar to Fountain model called as a *Spiral model* as shown in Fig. 1.14. Spiral model also follows iterative approach in each phase.

The Spiral model involves a little bit of analysis, followed by a little bit of design, a little bit of implementation and a little bit of testing. A loop of the spiral goes through some or all of the Waterfall phases. The idea is that each loop produces an output and by repeatedly following all the activities such as planning, analysis, implementation and review the final solution is reached. Engineering phase shown in quadrant III of Figure 1.14 involves coding, testing and putting the solution into use.

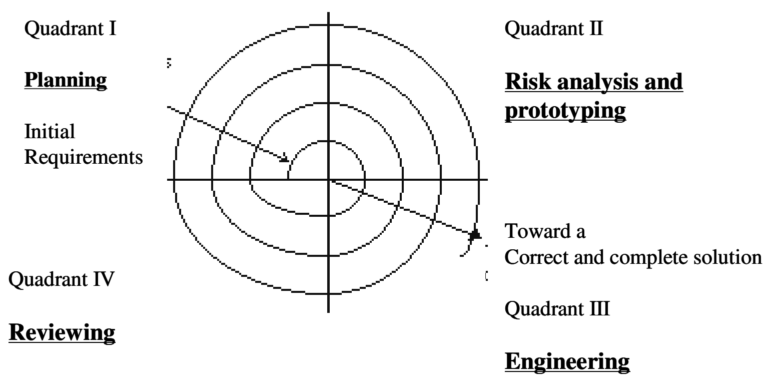

Figure 1.14: Spiral model

Both the Fountain model and Spiral model provided better solution for complex problems compared to top-down approach followed in the Waterfall model. The procedural and structured programming languages were found unsuitable for the bottom-up approach because a change in requirement, analysis, or design phase can cause the programming to start from the beginning once again. They lack flexibility, modifiability and software component reuse.

1.10 Evolution of a New Paradigm

The complexity of software required a change in the style of programming. It was aimed to:

1. produce reliable software
2. reduce production cost
3. develop reusable software modules
4. reduce maintenance cost

5. quicken the completion time of software development

The *Object-oriented model* was evolved for solving complex problems. It resulted in *object-oriented programming paradigms*. Object-oriented software development started in the 1980s. Object-oriented programming (OOP) seems to be effective in solving the complex problems faced by software industries. The end-users as well as the software professionals are benefited by OOP. OOP provides a consistent means of communication among analysts, designers, programmers and end users.

Object-oriented programming paradigm suggests new ways of thinking for finding a solution to a problem. Hence the programmers should keep their mind tuned in such a manner that they are not to be blocked by their preconceptions experienced in other programming languages such as structured programming. Proficiency in object-oriented programming requires talent, creativity, intelligence, logical thinking and the ability to build and use abstractions and experience.

If procedures or functions are considered as verbs and data items are considered as nouns, a procedure oriented program is organized around verbs while an object-oriented program is organized around nouns.

1.11 Natural Way of Solving a Problem

People tackle a number of problems in everyday life. It is very important to understand the way a problem is addressed. Consider a situation in an office.

> Manager wants to go to a customer's site. He wants to sign a letter before he leaves.

How does the manager solve this problem? The way by which the problem is addressed is shown in Figure 1.15.

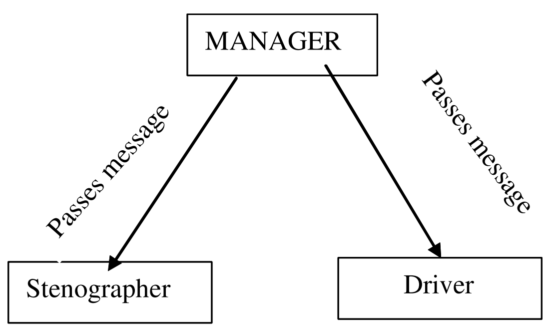

Figure 1.15: Message passing

The manager first calls the stenographer to prepare the letter and dictates the matter. The stenographer takes shorthand notes of the dictation and prepares the letter using a computer and a printer. Now the letter is ready for signing and the manager signs it. Then the manager calls the driver to take him to the customer's site. The driver along with the manager reaches the destination with the help of a car.

The manager delegates the responsibility of typing and taking the printed output to the stenographer. The driver is entrusted with the responsibility of taking him to the customer's site.