Software Development and Object-Oriented Programming Paradigms

Chapter

1

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 object-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 would 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 a 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 the 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 the 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 the 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 it.

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 Fig. 1.1. It is the act of finding a solution to a problem. The formulation of a 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.

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 Fig. 1.2. It defines the sequence of actions that produces the end state from the start state.

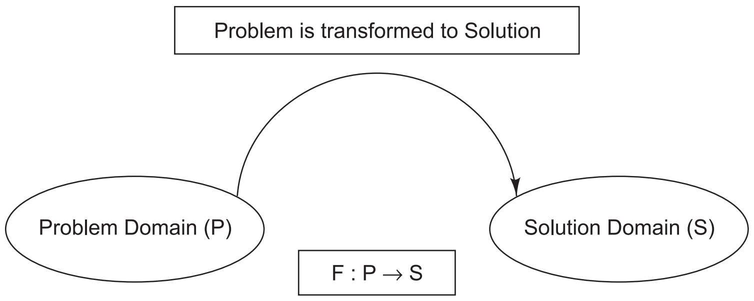

Fig. 1.1 Problem solving

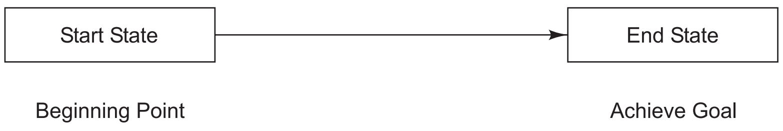

Fig. 1.2 Solution to a problem

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 the people associated with it.

1.3 TYPES OF PERSONS ASSOCIATED TO A SOLUTION

We may observe the three types of people associated with a solution to a problem as shown in Fig. 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.

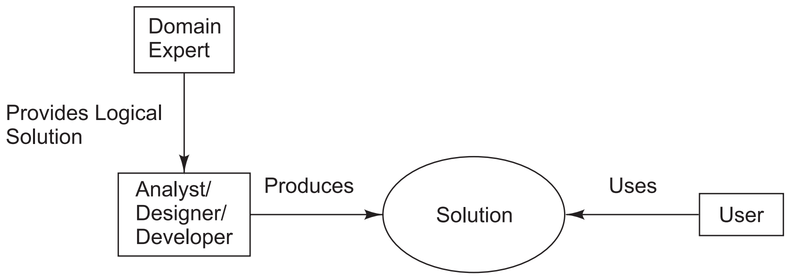

Fig. 1.3 People associated with the solution

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 a solution is quite challenging. The domain experts play a major role in formulating the solution. The formulation of a solution is important before writing a program. It requires logical thinking, careful planning, and a 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 Fig. 1.4.

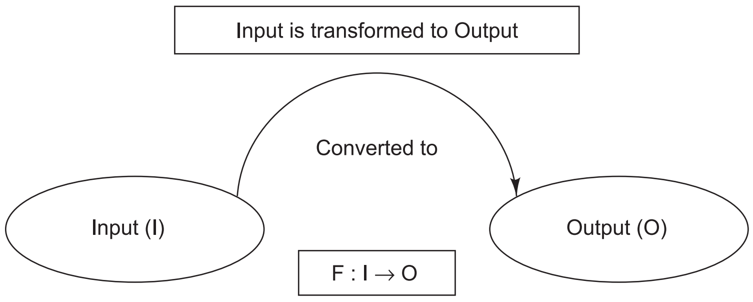

Fig. 1.4 Program

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 all 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 a known to an 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 a 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 Fig. 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 the solution domain by representing the operations/operators in the programming language that can be performed in the computer.

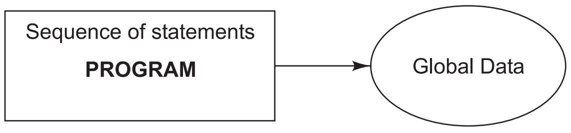

Fig. 1.5 Structure of the first generation languages

Second generation programming languages (1959–1961) introduced subprograms (functions or procedures or subroutines) as shown in Fig. 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.

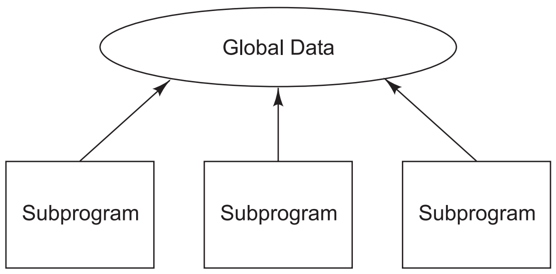

Fig. 1.6 Sturucture of the second generation languages

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 has only partially succeeded. Here also, the program is closer to the 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 Fig. 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 to 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.

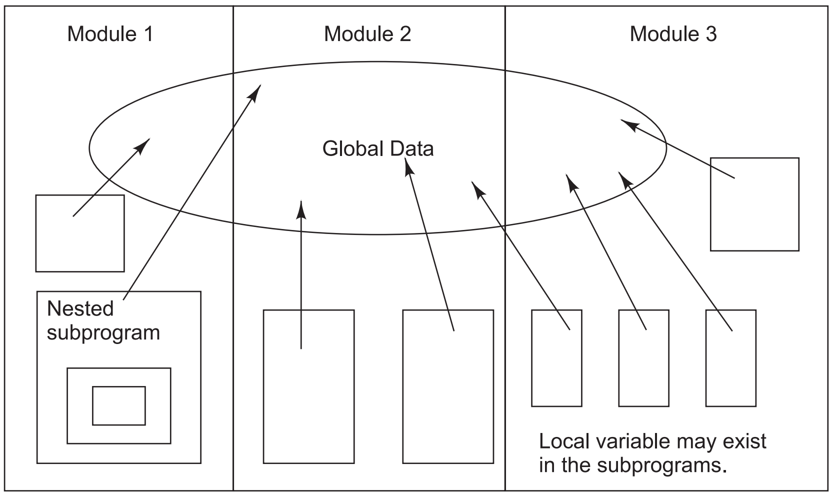

Fig. 1.7 Data in third generation programming languages

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.

[1] A scope identifies the portion of the 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.

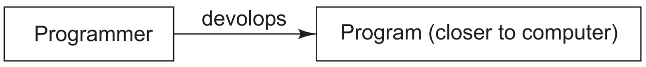

Fig. 1.8 Relationship between a program and programmer

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 Fig. 1.8. *Hence structured programming paradigms depend on the solution domain and not on the problem domain*. The data is not given importance regarding access permission.

To solve a complex problem using the *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 descriptions, always maintaining correctness at each level. At each step, either a control or a data structure is refined. Thus the *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 Fig. 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.

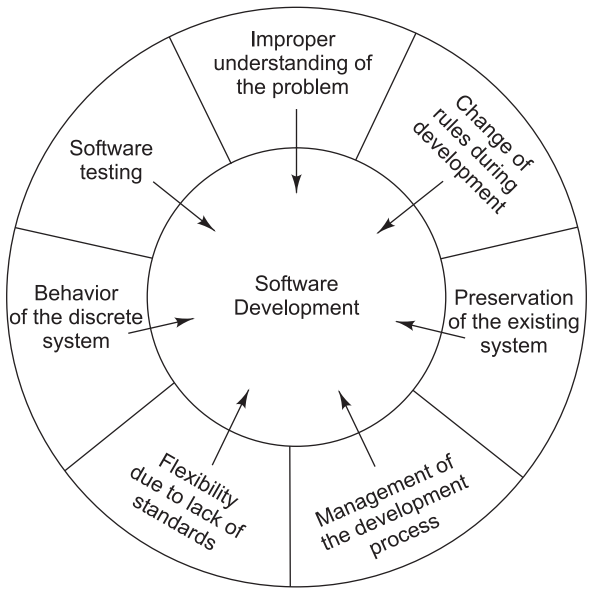
Improper understanding of the problem
Change of rules during development
Preservation of the existing system
Management of the development process
Flexibility due to lack of standards
Behavior of the discrete system
Software testing
Software Development

Fig. 1.9 Factors influencing software complexity

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 nonavailability 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.

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 the budget in the case of large and complex projects. 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 Fig. 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 the Y2K (Year 2000) problem in software applications. As a result, Indian software engineers have gained worldwide popularity, which has in turn led to rapid growth of IT industries in India.

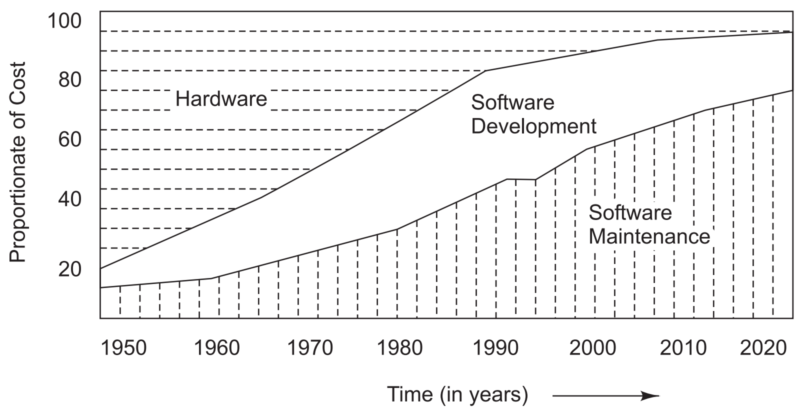

Fig. 1.10 System development cost

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 Fig. 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.

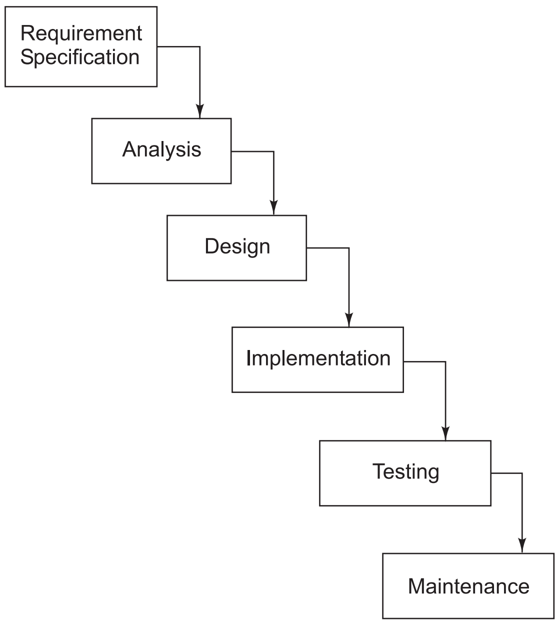

Fig. 1.11 Software development activities (Waterfall model)

Percentage of costs incurred during the different phases of SDLC is shown in Fig. 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 the 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 Fig. 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.

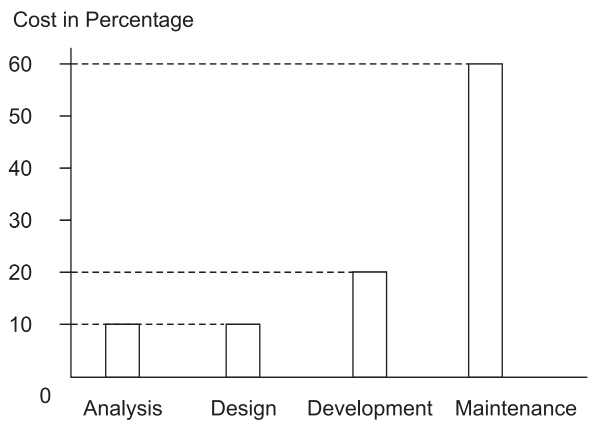

Fig. 1.12 Costs involved in SDLC

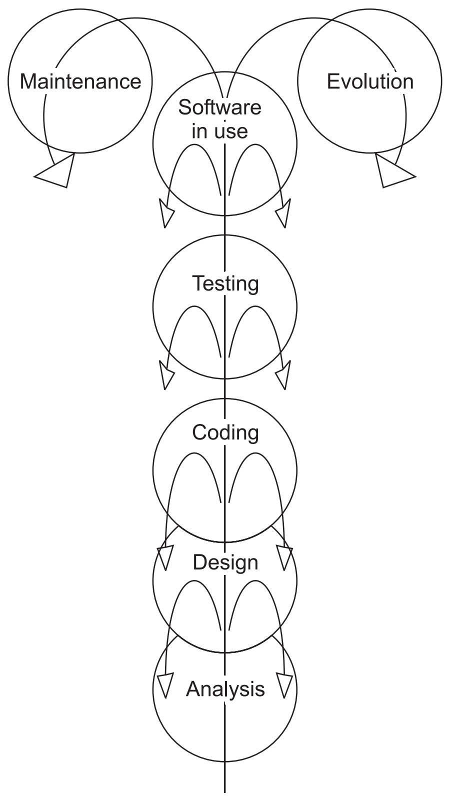

Fig. 1.13 Fountain model

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 a Fountain model called a *Spiral model*, as shown in Fig. 1.14. Spiral model also follows an 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 Fig. 1.14 involves coding, testing, and putting the solution into use.

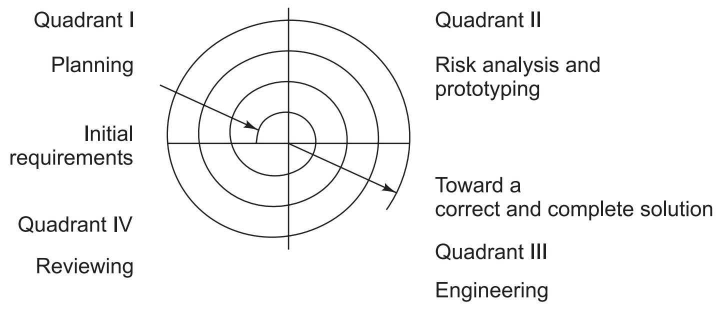

Fig. 1.14 Spiral model

Both the Fountain model and the Spiral model provided better solutions for complex problems compared to the 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 minds 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 Fig. 1.15.

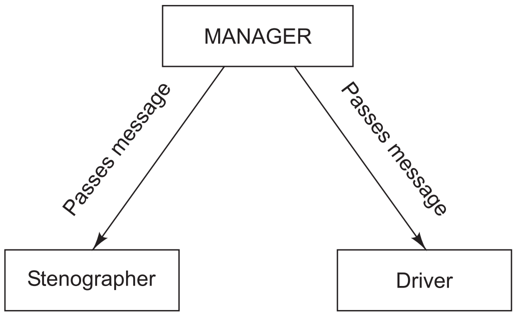

Fig. 1.15 Message passing

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. Thus, the manager uses two persons to complete the task. He doesn't bother to know how the stenographer prepares the document. By delegating the responsibility to someone, the manager is free from that work. The specific tasks assigned to the steno and to the driver are done independently. The stenographer makes use of another object (computer and printer or typewriter) to complete the task. The driver uses the car to go to the destination. The manager is able to perform the complex task by delegating the responsibilities to the concerned persons. Action is initiated by sending a message to the person responsible for the action. The message-receiving person accepts the responsibility and the task is carried out by means of a method. Thus, messages and methods play important roles in solving real-world problems. Message passing is the first principle to initiate an action by means of a method. Observe the responsibility-driven technique used in problem solving. Message passing resembles a *function call* in a structured programming language. A function is called to perform an action by passing parameters. Both message passing and function call result in performing a task. But there are differences between them. The differences between function call and message passing, shown in Table 1.1, must be understood before learning OOP.

Table 1.1 Comparison of function call and message passing

Function Call	Message Passing
1. Function call may use zero or more arguments.	Message passing uses at least one argument that identifies the receiving agent.
2. It always identifies a single piece of executable code.	The function name is called message selector. The same name may be associated with different receiving agents.
3. It is applied to data to carry out a task.	Message passing is a way to access the data. Message may invoke a function defined for a specific purpose.
4. Consumer is responsible for choosing functions that operate properly on the data.	Supplier is responsible for choosing the appropriate message.
5. There is no designated receiver in the function call.	There is a designated receiving agent in message passing.

If the way of solving a problem is viewed in depth, the concept of abstraction can be understood.

1.12 ABSTRACTION

The abstract view of solving a problem is an essential requirement as we do in a real-world problem. Consider the previous example of the situation in an office. The manager passes the information about the place of destination to the driver who performs the action of moving from the office to the desired site. The manager must know the person who is capable of doing this task even though he may not know driving. The driver takes care of the execution part of driving. In the perspective of the manager, the driver is an employee who knows driving and can take him to the desired place. This abstract information about the driver is enough for the manager. The manager is an officer employed in the office. For the driver the details of the officer like name and designation are enough. This is the abstract information about the officer. The driver uses a car to perform the task. In the perspective of a driver the features of a car are shown in Fig. 1.16. In the perspective of the manager, the type of car such as A/C or non-A/C and brand name may be important. Thus the abstract information of the same entity differs from individual to individual.

The essential features of an entity are known as *abstraction*. A feature may be either an attribute reflecting a property (or state or data) or an operation reflecting a method (or behavior or function). The features such as things in the trunk of a car, the medical history of the manager traveling in the car, and the working mechanism of the car engine are not necessary for the driver. The essential features of an entity in the perspective of the user define abstraction. A good abstraction is achieved by having:

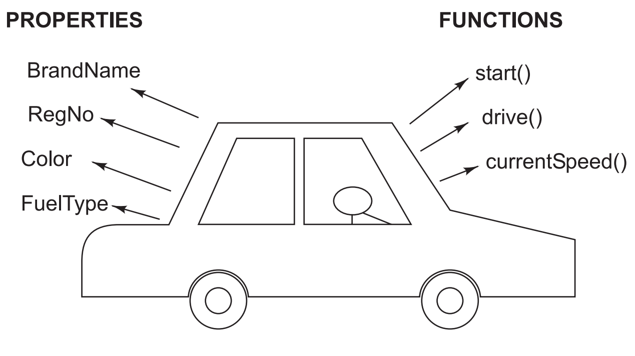

Fig. 1.16 Features of a car in the perspective of a driver

- meaningful name such as driver reflecting the function
- minimum and at the same time complete features
- coherent features

Abstraction specifies necessary and sufficient descriptions rather than implementation details. It results in separation of interface and implementation. The concepts of interface and implementation are discussed next.

1.13 INTERFACE AND IMPLEMENTATION

It is very important to know the difference between interface and implementation. For example, when a driver drives the car, he uses the steering to turn the car. The purpose of the steering is known very well to the driver, but the driver need not to know the internal mechanisms of different joints and links of various components connected to the steering.

An interface is the user's view of what can be done with an entity. It tells the user what can be performed. Implementation takes care of the internal operations of an interface that need not be known to the user, as shown in Fig. 1.17. The implementation concentrates on how an entity works internally. Their comparison is shown in Table 1.2.

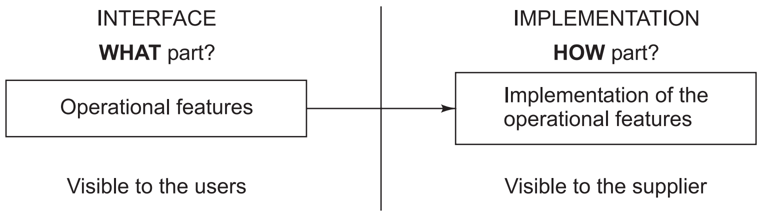

Fig. 1.17 Separation of interface from implementation

Table 1.2 Comparison of interface and implementation

Interface	Implementation
It is user's viewpoint. (**What** part)	It is supplier's viewpoint. (**How** part)
It is used to interact with the outside world.	It describes how the delegated responsibility is carried out.
User is permitted to access the interfaces only.	Functions or methods are permitted to access the data. Thus, supplier is capable of accessing data and interfaces.
It encapsulates the knowledge about the object.	It provides the restriction of access to data by the user.

1.14 ENCAPSULATION

From the user's point of view, a number of features are packaged in a capsule to form an entity. This entity offers a number of services in the form of interfaces by hiding the implementation details. The term *encapsulation* is used to describe the hiding of the implementation details. The advantages of encapsulation are:

- information hiding
- implementation independence

If the implementation details are not known to the user, it is called information hiding. Restriction of external access to features results in data hiding. The driver may not know the steering mechanism, but knows how to use it. Here, the hidden steering mechanism refers to information hiding. Whatever type of steering is used, the way of using the steering is the same. Rotating the steering wheel is an example of interface. The steering wheel is visible to the driver (user) and its function is not affected by the change in the implementation by a different type of steering mechanism such as power steering. The user's interface is not affected by changing the implementation mechanism. A change in the implementation is done easily without affecting the interface. This leads to implementation independence. Thus, the natural way of solving a problem involves abstraction and encapsulation. Conventional programming, which uses structured programming, is different from the natural way of solving a problem.

1.15 COMPARISON OF NATURAL AND CONVENTIONAL PROGRAMMING METHODS

In conventional programming, structured or procedural languages are used. In the structured programming approach, functions are defined according to the algorithm to solve the problem. Here, function abstractions are concentrated. A function is applied to some data to perform the actions on data. This approach may be called a *data-driven* approach, which involves operator/operand concept. It depends on the solution domain because the algorithm (solution) is closer to the coding of the program. The relationship between the programmer and the program is emphasized in the data-driven approach. The solution is solution-domain specific. Conventional programming follows the following principles:

- operator-operand concept
- function abstraction
- separation of data and functions

The development of the algorithm is given prime importance in conventional programming. The importance of data is not considered and hence, sometimes critical data having global access may result in miserable output. The abstraction followed is function abstraction and not data abstraction. Data and functionalities are considered as two separate parts.

But, in the natural way of solving real-world problems, the responsibility is delegated to an agent. The solution is proposed instead of developing an algorithm. The problem is solved by having a number of agents (interfaces). The interface part is the user's viewpoint, and hence the solution is not closer to the coding of the program. The real-world problem is solved using a responsibility-driven approach. In this approach, the relationship between the user and the programmer is emphasized. Here, the solution is problem domain-specific. The natural way of problem solving follows the following basic principles:

- message passing
- abstraction
- encapsulation

The importance of data is realized through object-oriented technology, which follows the natural way of solving problems. Data abstraction and data encapsulation help to make the abstract view of the solution with information hiding. Data is given the proper importance and action is initiated by message passing. Data and functionalities are put together resulting in objects and a collection of interacting objects are used to solve the problem. Object-oriented programming languages are developed based on object-oriented technology.

1.16 OBJECT-ORIENTED PROGRAMMING PARADIGMS

The object-oriented approach to programming is an easy way to master the management and complexity in developing software systems that take advantage of the strengths of data abstraction. Data-driven methods of programming provide a disciplined approach to the problems of data abstraction, resulting in the development of object-based languages that support only data abstraction. These object-based languages do not support the features of the object-oriented paradigm, such as inheritance or polymorphism. Depending on the object features supported, there are two categories of object languages:

1. Object-Based Programming Languages
2. Object-Oriented Programming Languages

Object-based programming languages support encapsulation and object identity (unique property to differentiate it from other objects) without supporting important features of OOP languages such as polymorphism, inheritance, and message based communication, although these features may be emulated to some extent. Ada, C, and Haskell are three examples of typical object-based programming languages.

Object-based language = Encapsulation + Object Identity

Object-oriented languages incorporate all the features of object-based programming languages, alongwith inheritance and polymorphism (discussed later in this chapter). Therefore, an object-oriented programming language is defined by the following statement:

Object-oriented language = Object-based features + Inheritance + Polymorphism

Object-oriented programming languages for projects of any size use *modules* to represent the physical building blocks of these languages. A module is a logical grouping of related declarations, such as objects or procedures, and replaces the traditional concept of *subprograms* that existed in earlier languages.

The following are important features in object-oriented programming and design:

1. Improvement over the structured programming paradigm.
2. Emphasis on data rather than algorithms.
3. Procedural abstraction is complemented by data abstraction.
4. Data and associated operations are unified, grouping objects with common attributes, operations, and semantics.

Programs are designed around the data on which it is being operated, rather than the operations themselves. Decomposition, rather than being algorithmic, is data-centric. Clear understanding of classes and objects are essential for learning object-oriented development. The concepts of classes and objects help in the understanding of object model and realizing its importance in solving complex problems.

Object-oriented technology is built upon *object models*. An *Object* is anything having crisply defined conceptual boundaries. Book, pen, train, employee, student, machine, etc., are examples of objects. But the entities that do not have crisply defined boundaries are not objects. Beauty, river, sky, etc., are not objects. *Model* is the description of a specific view of a real-world *problem domain* showing those aspects, which are considered to be important to the observer (user) of the problem domain. Object-oriented programming language directly influences the way in which we view the world. It uses the programming paradigm to address the problems in everyday life. It addresses the solution closer to the problem domain. *Object model is defined by means of classes and objects*. The development of programs using object model is known as object-oriented development.

To learn object-oriented programming concepts, it is very important to view the problem from the user's perspective and model the solution using object model.

1.17 CLASSES AND OBJECTS

The concepts of object-oriented technology must be represented in object-oriented programming languages. Only then, complex problems can be solved in the same manner as they are solved in real-world situations. OOP languages use classes and objects for representing the concepts of abstraction and encapsulation. The mapping of abstraction to a program is shown in Fig. 1.18.

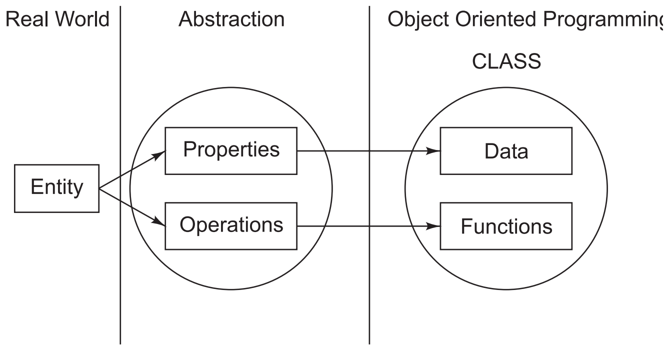

Fig. 1.18 Mapping real world entity to object oriented programming

The *software structure* that supports *data abstraction* is known as *class*. A class is a *data type* capturing the essence of an abstraction. It is characterized by a number of features. The class is a *prototype* or *blue print* or *model* that defines different features. A feature may be a data or an operation. Data are represented by *instance variables* or *data variables* in a class. The operations are also known as behaviors, or methods, or functions. They are represented by member functions of a class in C++ and methods in Java and C#.

A class is a data type and hence it cannot be directly manipulated. It describes a set of objects. For example,

apple is a fruit

implies that apple is an example of fruit. The term "fruit" is a type of food and apple is an instance of fruit. Likewise, a class is a type of data (data type) and object is an instance of class.

Similarly *car* represents a *class* (a model of vehicle) and there are a number of instances of car. Each instance of car is an object and the class car does not physically mean a car. An object is also known as *class variable* because it is created by the class data type. Actually, each object in an object-oriented system corresponds to a *real-world thing*, which may be a person, or a product, or an entity. The differences between class and object are given in Table 1.3.

Table 1.3 Comparison of Class and Object

Class	Object
Class is a data type.	Object is an instance of class data type.
It generates object.	It gives life to a class.
It is the prototype or model.	It is a container for storing its features.
Does not occupy memory location.	It occupies memory location.
It cannot be manipulated because it is not available in the memory.	It can be manipulated.

Instantiation of an object is defined as the process of creating an object of a particular class.

An object has:

- states or properties
- operations
- identity

Properties maintain the internal state of an object. Operations provide the appropriate functionality to the object. Identity differentiates one object from the other. Object name is used to identify the object. Hence, object name itself is an identity. Sometimes, the object name is mixed with a property to differentiate two objects. For example, differentiation of two similar types of cars, say MARUTI 800 may be differentiated by colors. If colors are also same, the registration number is used. Unique identity is important and hence the property reflecting unique identity must be used in an object.

The properties of an object are important because the outcome of the functions depends on these properties. The functions control the properties of an object. They act and react to messages. The message may cause a change in the property of an object. Thus, the behavior of an object depends on the properties. For example, assume a property called brake condition for the class car. If the brake is not in working condition, guess the behavior of car. The outcome may be unexpected.

Similarly, in a student mark statement, the `result()` behavior depends on the data called `marks`. The property of `resultStatus` may be modified based on the marks.

1.18 FEATURES OF OBJECT-ORIENTED PROGRAMMING

The fundamental features of object-oriented programming are as follows:

- Encapsulation
- Data Abstraction
- Inheritance
- Polymorphism
- Extensibility
- Persistence
- Delegation
- Genericity
- Object Concurrency
- Event Handling
- Multiple Inheritance
- Message Passing

A model of these features and the way they relate to the Java language is shown in Fig. 1.19.

1.18.1 Encapsulation

The process, or mechanism, by which you combine code and the data it manipulates into a single unit, is commonly referred to as *encapsulation*. Encapsulation provides a layer of security around manipulated data, protecting it from external interference and misuse. In Java, this is supported by *classes* and *objects*.

1.18.2 Data Abstraction

Real-world objects are very complex and it is very difficult to capture the complete details. Hence, OOP uses the concepts of abstraction and encapsulation. Abstraction is a design technique that focuses on the essential attributes and behavior. It is a named collection of essential attributes and behavior relevant to programming a given entity for a specific problem domain, relative to the perspective of the user.

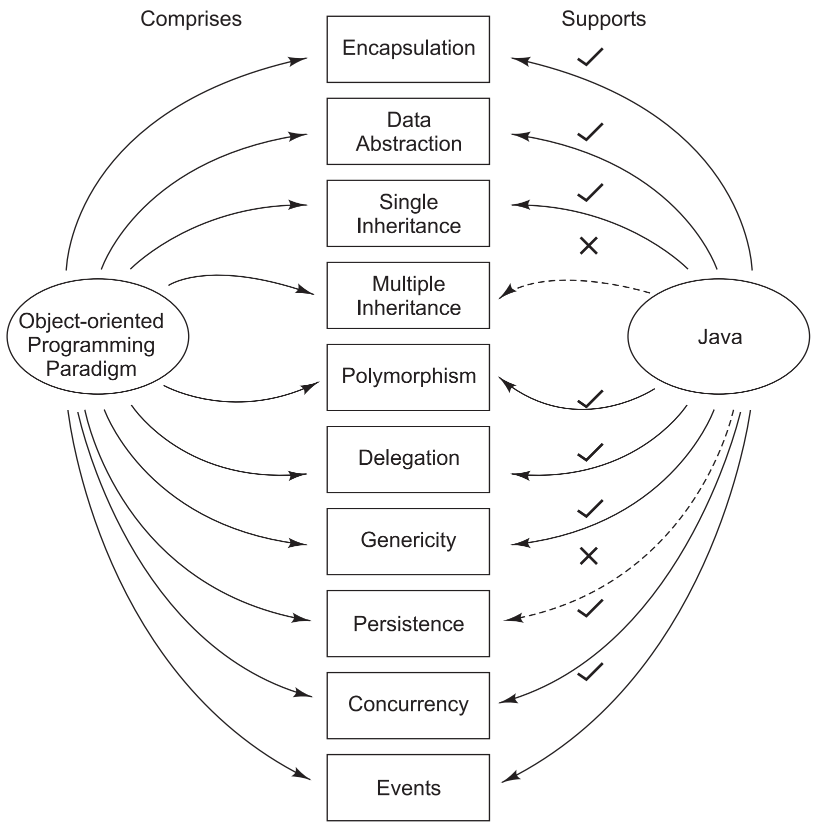

Fig. 1.19 Features of the object-oriented paradigm

Closely related to encapsulation, data abstraction provides the ability to create user-defined data types. Data abstraction is the process of *abstracting* common features from objects and procedures, and creating a single interface to complete multiple tasks. For example, a programmer may note that a function that prints a document exists in many classes, and may *abstract* that function, creating a separate class that handles any kind of printing. Data abstraction also allows user-defined data types that, while having the properties of built-in data types, it also allows a set of permissible operators that may not be available in the initial data type. In Java, the *class* construct is used for creating user-defined data types, called Abstract Data Types (ADTs).

A good abstraction is characterized by the following properties:

1. *Meaningful way of naming* An abstraction must be named in a meaningful way. The name itself must reflect the attributes and behaviors of the object for which the abstraction is made.
2. *Minimum features* An abstraction must have only essential attributes and behaviors, no more and no less.
3. *Complete details*
4. *Coherence*

An abstraction should define a related set of attributes and behavior to satisfy the requirement. Knowing the ISBN number of a book is irrelevant for a reader whereas for a librarian, it is very important for classification. Hence, the abstraction must be relevant to the given application.

Separation of interface and implementation is an abstraction mechanism in object-oriented programming language. Separation is useful in simplifying a complex system. It refers to distinguishing between a goal and a plan. It can be stated as separating "what" is to be done from "how" it is to be done. The separation may be well understood by the following equivalent terms, as shown in Table 1.4.

Table 1.4 Equivalent terms reflecting separation

What	How
Goals	Plans
Policy	Mechanism
Interface/requirement	Implementation

The implementation is hidden and it is important only for the developer. Separation in software design is an important concept for simplifying the development of software. Also, separation provides flexibility in the implementation. Several implementations are possible for the same interface. Sometimes, a single implementation can satisfy several interfaces.

Encapsulation is a process of hiding nonessential details of an object. It allows an object to supply only the requested information to another object and hides nonessential information. Since it packages data and methods of an object, an implicit protection from external tampering prevails. However, an entire application cannot be hidden. A part of the application needs to be accessed by users to use an application. Abstraction is used to provide access to a specific part of an application. It provides access to a specific part of data, while encapsulation hides data.

Rendering abstraction in software is an implicit goal of programming. Object-oriented programming languages permit abstractions to be represented more easily and explicitly. Object-oriented programming languages use classes and objects for representing abstractions. A class defines the specific structure of a given abstraction. It has a unique name that conveys the meaning of the abstraction. Class definition defines the common structure once. It allows "reuse" when creating new objects of the defined structure. An object's properties are exactly those described by its class. Two main parts of an object are:

- *Interface*: The user's view of the operations performed by an object is known as the interface part of that object.
- *Implementation*: The implementation of an object describes how the entrusted responsibility in the interface is achieved.

It is important to observe abstraction from the perspective of the user. Software is developed for end-users. Hence, the abstraction is captured from the user's point of view. For the same reason abstraction varies from viewer to viewer. For example, a book abstraction viewed by a librarian is different from the abstraction viewed by a reader of the book. A librarian may consider the following features:

Attributes	**Functions**
`title`	`printBook( )`
`author`	`getDetails( )`
`publisher`	`sortTitle( )`
`cost`	`sortAuthor( )`
`accNumber`	
`ISBNnumber`	

A reader may consider the following features:

Attributes	**Functions**
`title`	`bookDetails()`
`author`	`availability()`
`content`	`tokenDetails()`
`examples`	
`exercises`	
`index`	

Here, the attributes are data and the functions are operations or behaviors related to data. If an application software is to be developed for a library, the abstraction captured by the librarian is important. The reader's point of view is not necessary. Thus abstraction differs from viewer to viewer. Abstraction relative to the perspective of the user is very important in software development.

A simple view of an object is a combination of properties and behavior. The method name with arguments represents the interface of an object. The interface is used to interact with the outside world. Object-oriented programming is a packaging technology. Objects encapsulate data and behavior hiding the details of implementation. The concept of implementation hiding is also known as *information hiding*. Since data is important, the users cannot access this data directly. Only the interfaces (methods) can access or modify the encapsulated data. Thus, *data hiding* is also achieved. The restriction of access to data within an object to only those methods defined by the object's class is known as encapsulation. Also, implementation is independently done improving software reuse concept. Interface encapsulates knowledge about the object. Encapsulation is an abstract concept. Table 1.5 gives a clear picture about the different concepts.

Table 1.5 Comparison of Abstraction and Encapsulation

Abstraction	Encapsulation
Abstraction separates interface and implementation.	Encapsulation groups related concepts into one item.
User knows only the interfaces of the object and how to use them according to abstraction. Thus, it provides access to a specific part of data.	Encapsulation hides data and the user cannot access the same directly (data hiding).
Abstraction gives the coherent picture of what the user wants to know. The degree of relatedness of an encapsulated unit is defined as cohesion. High cohesion is achieved by means of good abstraction.	Coupling means dependency. Good systems have low coupling. Encapsulation results in lesser dependencies of one object on other objects in a system that has low coupling. Low coupling may be achieved by designing a good encapsulation.
Abstraction is defined as a data type called class which separates interface from implementation.	Encapsulation packages data and functionality and hides the implementation details (information hiding).
The ability to encapsulate and isolate design from execution information is known as abstraction.	Encapsulation is a concept embedded in abstraction.

Classes and objects represent abstractions in OOP languages. Class is a common representation with definite attributes and operations having a unique name. Class can be viewed as a user-defined data type. Data types cannot be used in a program for direct manipulation. A variable of a particular data type is defined first as a container for storage. The variables are manipulated after holding data in them. For example,

```
int year, mark ;
```

is a declaration of variables in C. This statement conveys to the compiler that *year* and *mark* are instances of integer data type. Likewise, in OOP, a class is a data type. A variable of a class data type is known as an object. An object is defined as an instance of a class. For example, if *Book* is a defined class,

```
Book cBook, javaBook ;
```

declares the variables `cBook` and `javaBook` of the `Book` class type. Thus, classes are software prototypes for objects. Creation of a class variable or an object is known as *instantiation* (creation of an instance of a class). The objects must be allocated in memory. Classes cannot be allocated in memory.

1.18.3 Inheritance

Inheritance allows the extension and reuse of existing code, without having to repeat or rewrite the code from scratch. Inheritance involves the creation of new classes, also called *derived* classes, from existing classes (*base* classes). Allowing the creation of new classes enables the existence of a hierarchy of classes that simulates the class and subclass concept of the real world. The new derived class inherits the members of the base class and also adds its own. For example, a banking system would expect to have customers, of which we keep information such as name, address, etc. A subclass of customer could be customers who are students, where not only we keep their name and address, but we also track the educational institution they are enrolled in.

Inheritance is mostly useful for two programming strategies: extension and specialization. Extension uses inheritance to develop new classes from existing ones by adding new features. Specialization makes use of inheritance to refine the behavior of a general class.

1.18.4 Multiple Inheritance

When a class is derived through inheriting one or more base classes, it is being supported by *multiple inheritance*. Instances of classes using multiple inheritance have instance variables for each of the inherited base classes. Java does not support multiple inheritance. However, Java allows any class to implement multiple interfaces, which to some extent appears like a realization of multiple inheritance.

1.18.5 Polymorphism

Polymorphism allows an object to be processed differently by data types and/or data classes. More precisely, it is the ability for different objects to respond to the same message in different ways. It allows a single name or operator to be associated with different operations, depending on the type of data it has passed, and gives the ability to redefine a *method* within a *derived class*. For example, given the *student* and *business* subclasses of *customer* in a banking system, a programmer would be able to define different *getInterestRate()* methods in *student* and *business* to override the default interest *getInterestRate()* that is held in the *customer* class. While Java supports method overloading, it does not support operator overloading.

1.18.6 Delegation

Delegation is an alternative to class inheritance. Delegation allows an object composition to be as powerful as inheritance. In delegation, two objects are involved in handling a request: methods can be delegated by one object to another, but the *receiver* stays bound to the object doing the delegating, rather than the object being delegated to. This is analogous to child classes sending requests to parent classes. In Java, delegation is supported as more of a *message forwarding* concept.

1.18.7 Genericity

Genericity is a technique for defining software components that have more than one interpretation depending on the data type of parameters. Thus, it allows the abstraction of data items without specifying their exact type. These unknown (generic) data types are resolved at the time of their usage (e.g., through a function call), and are based on the data type of parameters. For example, a *sort* function can be parameterized by the type of elements it sorts. To invoke the parameterized *sort()*, just supply the required data type parameters to it and the compiler will take care of issues such as creation of actual functions and invoking that transparently. Genericity is introduced in Java 1.5, implemented as generic interfaces that take *parameter* types.

1.18.8 Persistence

Persistence is the concept by which an object (a set of data) outlives the life of the program, existing between executions. All database systems support persistence, but, persistence is not supported in Java. However, persistence can be simulated through use of *file streams* that are stored on the file system.

1.18.9 Concurrency

Concurrency is represented in Java through threading, synchronization, and scheduling. Using concurrency allows additional complexity to the development of applications, allowing more flexibility in software applications.

1.18.10 Events

An event can be considered a kind of interrupt: they interrupt your program and allow your program to respond appropriately. In a conventional, nonobject-oriented language, processing proceeds literally through the code: code is executed in a 'top-down' manner. The flow of code in a conventional language can only be interrupted by loops, functions, or iterative conditional statements. In an object-oriented language such as Java, events interrupt the normal flow of program execution. Objects can pass information and control from themselves to another object, which in turn can pass control to other objects, and so on. In Java, events are handled through the *EventHandler* class, which supports dynamically generated listeners. Java also implements event functionality in classes such as the *Error* subclass: abnormal conditions are *caught* and *thrown* so they can be handled appropriately.

1.19 MODULARITY

The complexity of a program can be reduced by partitioning the program into individual modules. In object-oriented programming languages, classes and objects form the logical structure of a system. Modules serve as the physical containers in which the classes and objects are declared. Modularity is the property of a system that has been decomposed into a set of cohesive and loosely coupled modules. A module is an indivisible unit of software that can be reused. The boundaries of modules are established to minimize the interfaces among different parts of the development organization. Modules are frequently used as an implementation technique for abstract data type. Abstract data type is a theoretical concept and module is an implementation technique. Each class is considered to be a module in OOP.

The responsibilities of classes are defined by means of their attributes and behavior. But a single object alone is not very useful. Higher order functionality and complex behavior are achieved through interaction of objects in different modules. Hence, interaction of objects is very important. Software objects interact and communicate with each other by sending messages to each other.

The activities are initiated by the transmission of a message to an object responsible for the action. The message encodes the request and the information is passed along with the message as parameters. There are three components to comprise a message:

- The receiver objects to whom the message is addressed.
- The name of the function performing the action.
- The parameters required by the function.

Interaction between objects is possible with the help of message passing. In the case of distributed applications, objects in different machines can also send and receive messages.

1.20 HOW TO DESIGN A CLASS?

A class is designed with a specific goal. Its purpose must be clear to the users. An entity in solving a problem is categorized as a class if there is a need for more than one instance of this class. Also, it is very important to entrust a responsibility to an object. Presenting simply the behaviors such as reading data and displaying data in a class is a poor design of a class. To perform complex tasks, one class must jointly work with the other classes to perform the task. This approach is known as collaboration among classes. The class must be designed with essential attributes and behavior to reflect an idea in the real world.

The terms class and object are very important in object-oriented programming. A class is a prototype or blueprint or model that defines the variables and functions in it. The variables defined in a class represent the *data*, or *states*, or *properties*, or *attributes* of a visible thing of a certain type.

Classes are user-defined data types. It is possible to create a lot of objects of a class. The important advantages of classes are:

- Modularity
- Information hiding
- Reusability

1.21 DESIGN STRATEGIES IN OOP

Object-oriented programming includes a number of powerful design strategies based on software engineering principles. Design strategies allow the programmers to develop complex systems in a manageable form. They have been evolved out of decades of software engineering experience. The basic design strategies embedded in object-oriented programming are:

i. Abstraction
ii. Composition
iii. Generalization

The existing object-oriented programming languages support most of these features.

Abstraction is clearly discussed in the Section 1.18.2.

1.21.1 Composition

A complex system is organized using a number of simpler systems. An organized collection of smaller components interacting to achieve a coherent and common behavior is known as composition. There are two types of composition:

1. Association
2. Aggregation

Aggregation considers the composed part as a single unit whereas association considers each part of composition as a separate unit. For example, a computer is an association of CPU, keyboard, and monitor. Each part is visible and manipulated by the user. CPU is an aggregation of processor memory and control unit. The individual parts are not visible and they cannot be manipulated by the user. Both types of composition are useful. Aggregation provides greater security because its structure is defined in advance and cannot be altered at runtime. Association offers greater flexibility because the relationships among the visible units can be redefined at run time. It adapts to changing conditions in its execution environment by replacing one or more of its components. The two types of composition are frequently used together. A computer is an example for combination of both association and aggregation.

1.21.2 Generalization

Generalization identifies the common properties and behaviors of abstractions. It is different from abstraction. Abstraction is aimed at simplifying the description of an entity, whereas generalization identifies commonalities among a set of abstractions. Generalizations are important since they are like "laws" or "theorems", which lay the foundation for many things. Generalization helps to develop software capturing the idea of similarity.

The different types of generalization are:

1. hierarchy
2. genericity
3. polymorphism
4. pattern

1. Hierarchy The first type of generalization uses a tree-structured form to organize commonalities. A generalization/specialization hierarchy is achieved with the help of inheritance in object-oriented programming languages. The advantages are:

- Knowledge representation in a particular form.
- The intermediate levels in the hierarchy provide the names that can be used among developers and between developers and application domain experts.
- A new specialization at any level can be extended.
- New attributes and behavior can be easily added.

2. Genericity It refers to a generic class, which is meant for accepting different types of parameters. A stack class can be considered as a generic class if it is capable of accepting integer data as well as float or double or char data also. This type of generalization is known as genericity.

3. Polymorphism The term *poly* means *many* and the term *morph* means to *form*. Then polymorphism concerns the possibility for a single property of exposing multiple possible states. The generally accepted definition for this term in object oriented programming is the capability of objects belonging to the same class hierarchy to react differently to the same method call. This means that a function may be defined in different forms with the same function name. It is possible to implement different functionalities using a common name for a function. Polymorphism provides a way of generalizing algorithms. Late binding or dynamic binding (discussed later) is required to implement polymorphism in object-oriented programming. Based on the parameters passed, the compiler dynamically identifies the function to be invoked and it is known as dynamic binding.

4. Pattern A pattern is a generalization of a solution for a common problem. An architecture or model is a large scale pattern used in computer science. Client-server model is an example of a large-scale pattern. A pattern is a distinct form of generalization. It gives a general form of solution. A pattern need not be expressed in code at all. The elements of the pattern are represented by classes. The relationships among the elements may be defined by association, aggregation, and/or hierarchy.

1.22 COMPARISON OF STRUCTURED AND OBJECT-ORIENTED PROGRAMMING

It is essential to understand the basic differences between structured programming and OOP concepts, which is shown in Table 1.6.

Table 1.6 Difference between Structured and OO Programming

Structured Programming	Object-Oriented Programming
Top-down approach is followed.	Bottom-up approach is followed.
Focus is on algorithm and control flow.	Focus is on object model.
Program is divided into a number of submodules, or functions, or procedures.	Program is organized by having a number of classes and objects.
Functions are independent of each other.	Each class is related in a hierarchical manner.
No designated receiver in the function call.	There is a designated receiver for each message passing.
Views data and functions as two separate entities.	Views data and function as a single entity.
Maintenance is costly.	Maintenance is relatively cheaper.
Software reuse is not possible.	Helps in software reuse.
Function call is used.	Message passing is used.
Function abstraction is used.	Data abstraction is used.
Algorithm is given importance.	Data is given importance.
Solution is solution-domain specific.	Solution is problem-domain specific.
No encapsulation. Data and functions are separate.	Encapsulation packages code and data altogether. Data and functionalities are put together in a single entity.
Relationship between programmer and program is emphasized.	Relationship between programmer and user is emphasized.
Data-driven technique is used.	Driven by delegation of responsibilities.

1.23 OBJECT-ORIENTED PROGRAMMING LANGUAGES

Several object-oriented programming languages have been invented since 1960. Some well-known ones are listed in Table 1.6. Among them, C++, Java, and C# are the three most commercially successful OOP languages. Inventors and features of various OOP languages are given in Table 1.7.

Simula Simula was the first object-oriented language with syntax similar to Algol. Concurrent processes are managed by scheduler class. This language is best suited to the simulation of parallel systems. It allows classes with attributes and procedures that are public by default. It is also possible to declare them as private. Inheritance and virtual functions are supported. Memory is managed automatically with garbage collection.

Table 1.7 OO Programming languages
(a) OO programming languages and their inventors

Language	Inventor, Year	Organization
Simula	Kristen Nygaard and Ole-Johan Dahl, 1960	Norwegian Defense Research Establishment, Norway
Ada	Jean Ichbiah, 1970	Honeywell-CII-Bull, France
Smalltalk	Alan Kay, 1970	Xerox PARC, USA
C++	Bjarne Stroustrup, 1980	AT&T Bell Labs, USA
Objective C	Brad Cox, 1980	Stepstone, USA
Object Pascal	Larry Tesler, 1985	Apple Computer, USA
Eiffel	Bertrand Meyer, 1992	Eiffel Software, USA
Java	James Gosling, 1996	Sun Microsystems, USA
C#	Anders Hejlsberg, 2000	Microsoft, USA

(b) OO programming languages and comparison of their features

Feature	Java	C++	Smalltalk	Objective C	Simula	Ada	Eiffel	C#
Encapsulation	√	√	Poor	√	√	√	√	√
Single inheritance	√	√	√	√	√	X	√	√
Multiple inheritance	X	√	X	√	X	X	√	X
Polymorphism	√	√	√	√	√	√	√	√
Binding (early or late)	Late	Both	Late	Both	Both	Early	Early	Late
Concurrency	√	Poor	Poor	Poor	√	Difficult	√	√
Garbage collection	√	X	√	√	√	X	√	√
Persistent objects	X	X	promised	X	X	Like 3GL	Limited	X
Genericity	√	√	X	X	X	√	√	√
Class libraries	√	√	√	√	√	Limited	√	√

Ada Ada was developed by Jean Ichbiah and his team at Bull in the late 1970s. It was named after Augusta Ada, daughter of Byron, the famous romantic poet. It is a general-purpose language. An abstract type is implemented as a package in Ada. Each package can contain abstract types. The concept of genericity is introduced at the level of types and packages.

Smalltalk It was designed by Alan Kay at Xerox PARC during the 1970s. It is a general purpose language. It allows polymorphism. Automatic garbage collection is provided. Generalization of the object concept is another original contribution from Smalltalk.

C++ C++ was designed by Bjarne Stroustrup in the AT and T Bell Laboratories in the early 1980s. It borrowed the concepts of class, subclass, inheritance and polymorphism from Simula. The name C++ was coined by Rick Mascitti in 1983.

Objective C It is a general-purpose language designed by B. Cox. It extends C with an object model based on Smalltalk 80. It does not support metaclasses, which are classes used to describe other classes. Simple inheritance is supported. There are generic classes and no memory garbage collector.

Object Pascal It is an extension of Pascal, developed by Apple for the Macintosh in the early 1980s. Simple inheritance dynamic binding is supported. There is no automatic garbage collection.

Eiffel It was developed by Bertrand Meyer in 1992 for both scientific and commercial applications. Exception management is a feature supported by this language.

Java It is a pure object-oriented language developed by Arnold and Gosling in 1996. It helps in developing small applications called applets which can be integrated into web pages. It supports multithreading. It supports encapsulation, inheritance, polymorphism, genericity, and dynamic binding.

C# It is an object-oriented programming language developed by Microsoft Corporation for its new .NET Framework. It is derived from C and C++; appears very similar to Java. It supports encapsulation, inheritance, polimorphysm, genericity, and late binding.

1.24 REQUIREMENTS OF USING OOP APPROACH

The method of solving complex problems using OOP approach requires:
- Change in mindset of programmers, who are familiar with structured programming.
- Closer interaction between program developers and end-users.
- Much concentration on requirement, analysis, and design.
- More attention for system development than just programming.
- Intensive testing procedures.

1.25 ADVANTAGES OF OBJECT-ORIENTED PROGRAMMING

The following are the advantages of software developed using object-oriented programming:
1. Software reuse is enhanced.
2. Software maintenance cost can be reduced.
3. Data access is restricted providing better data security.
4. Software is easily developed for complex problems.
5. Software may be developed meeting the requirements on time, on the estimated budget.
6. Software has improved performance.
7. Software quality is improved.
8. Class hierarchies are helpful in the design process allowing increased extensibility.
9. Modularity is achieved.
10. Data abstraction is possible.

1.26 LIMITATIONS OF OBJECT-ORIENTED PROGRAMMING

1. The benefits of OOP may be realized after a long period.
2. Requires intensive testing procedures.
3. Solving a problem using OOP approach consumes more time than the time taken by structured programming approach.

1.27 APPLICATIONS OF OBJECT-ORIENTED PROGRAMMING

If there is complexity in software development, object-oriented programming is the best paradigm to solve the problem. The following areas make use of OOP:

1. Image processing
2. Pattern recognition
3. Computer assisted concurrent engineering
4. Computer aided design and manufacturing
5. Computer aided teaching
6. Intelligent systems
7. Data base management systems
8. Web based applications
9. Distributed computing and applications
10. Component based applications
11. Business process reengineering
12. Enterprise resource planning
13. Data security and management
14. Mobile computing
15. Data warehousing and data mining
16. Parallel computing

Object concept helps to translate our thoughts to a program. It provides a way of solving a problem in the same way as a human being perceives a real world problem and finds out the solution. It is possible to construct large reusable components using object-oriented techniques. Development of reusable components is rapidly growing in commercial software industries.

SUMMARY 1.28

Computers are used to solve problems. Different styles of programming have evolved in the history of generation of languages. But the problem of reuse and maintenance was not solved by those early languages and this led to the phenomenon called software crisis. To overcome the limitations, software engineering principles were applied and the object-oriented paradigm model was found to be suitable for addressing, modeling, and solving complex problems. The diffusion of this paradigm is the result of a continuous shift of programming abstractions from the solution domain to the problem domain. The more the problems to solve got complex the more we moved to models more close to the problem domain for solving them. From machine language to the object-oriented model and beyond. This constant movement has made the activity of finding a solution easier, more understandable, and maintainable.

Object-oriented programming uses object models and it resembles the natural way of solving a problem. The concepts of abstraction and encapsulation are used in OOP. The essential features of an entity are known as abstraction. Abstraction separates the interface from implementation. Encapsulation insulates the data by wrapping them up using methods. Classes and objects are the fundamental concepts that render abstractions. Classes are user-defined data types and objects are instances of a class. The features of OOP are discussed to realize the importance of OOP approach. The design strategies such as abstraction, composition, and generalization are embedded in OOP. Examples of OOP languages, advantages, and applications of OOP are presented to know the importance of OOP. Java is an OOP language. The history of development of Java and the runtime environment of Java are described in the next chapter.

1.29 EXERCISES

Objective Questions

1.1 Both Java and C# are _________ programming language.
1.2 Mapping of problem domain to solution domain is so called _________.
1.3 A program is a set of instructions written in a _________.
1.4 _________ approach general requires decomposing a complex problem into smaller problems.
1.5 _________ process is useful to make a software reliable.
1.6 _________ is a software development model that follows top-down approach.
1.7 The software structure that supports data abstraction is known as_________.
1.8 The process, or mechanism, by which you combine code and the data it manipulates into a single unit, is commonly referred to as _________ .
1.9 _________ allows an object to be processed differently by data types and/or data classes.
1.10 The basic design strategies embedded in object-oriented programming are: _________ , _________ and _________.
1.11 Bottom-up and top-down are the two very common problem solving strategies: True or False.
1.12 C and C++ are structural programming languages, Java and C# are object-oriented programming languages: True or False.
1.13 Class is a data type and Object is an instance of a class: True or False.
1.14 Abstract Data Types is a term referring to an abstract class: True or False.
1.15 Encapsulation hides the data and the user cannot access them directly: True or False.
1.16 Reusability is an important aspect of designing classes: True or False.
1.17 Object inheritance is a way of achieving genericity: True or False.
1.18 The focus of Object-Oriented Programming is the algorithm and control flow in a way of defining classes: True or False.
1.19 Structural programming language does not have a way of defining classes: True or False.
1.20 The testing is used to help Object-oriented programs instead of structural programs: True or False.

Review Questions

1.21 What is a problem domain?
1.22 What is a solution domain?
1.23 What is a programming paradigm?
1.24 What is the difference between monolithic programming and procedural programming?
1.25 What are the features of structured programming?
1.26 List the features of the programming languages in each generation.
1.27 What is top-down approach?
1.28 Explain the factors that influence the complexity of software development.
1.29 What are the factors that influence software crisis?
1.30 How is the software crisis avoided?
1.31 What are the important activities involved in Software development life cycle?
1.32 Explain the Waterfall model in software development.
1.33 What are the various costs involved in the Software development life cycle?

1.34 Explain the Fountain model related to software development.
1.35 What is bottom-up approach?
1.36 Explain the Spiral model related to software development.
1.37 Explain the concept of message passing.
1.38 Explain the concept of abstraction.
1.39 Compare interface and implementation.
1.40 Explain the concept of encapsulation.
1.41 What are the advantages of encapsulation?
1.42 Explain the basic principles involved in the natural way of problem solving.
1.43 What are the principles involved in conventional programming?
1.44 What are class and objects?
1.45 List properties and access methods for a typical student class?
1.46 Distinguish between a class and an object.
1.47 What are the features of an object?
1.48 Compare abstraction and encapsulation.
1.49 Explain the basic design strategies embedded in OOP.
1.50 What are the characteristics of a good abstraction?
1.51 Explain the concept of separation.
1.52 Explain the different types of generalization.
1.53 What is composition? Explain its types.
1.54 Compare structured programming and OOP.
1.55 State any five OOP Languages.
1.56 What are the advantages of OOP?
1.57 What are the limitations of OOP?
1.58 What are the important points to be considered while solving a problem using OOP approach?