C++

Genesis InSoft Limited

A name you can trust

1-7-1072/A, Opp. Saptagiri Theatre, RTC * Roads, Hyderabad - 500 020

Genesis Computers (A unit of Genesis InSoft Limited) started its operations on March 16th 1992 in Hyderabad, India primarily as a centre for advanced software education, development and consultancy.

Training is imparted through lectures supplemented with on-line demonstrations using audio visual aids. Seminars, workshops and demonstrations are organized periodically to keep the participants abreast of the latest technologies. Genesis InSoft Ltd. is also involved in software development and consultancy.

We have implemented projects/training in technologies ranging from client server applications to web based. Skilled in PowerBuilder, Windows C SDK, VC++, C++, C, Visual Basic, Java, J2EE, XML, WML, HTML, UML, Java Script, MS.NET (C#, VB.NET and ASP.NET, ADO.NET etc), PHP, Jhoomla, Zend Framework, JQuery, ExtJS, PERL, Python, TCL/TK etc. using Oracle, MySql and SQL Server as backend databases.

Genesis has earned a reputation of being in forefront on Technology and is ranked amongst the top training institutes in Hyderabad city. The highlight of Genesis is that new emerging technologies are absorbed quickly and applied in its areas of operation.

We have on our faculty a team of highly qualified and trained professionals who have worked both in India and abroad. So far we have trained about 51,000+ students who were mostly engineers and experienced computer professionals themselves.

Tapadia (MS, USA), the founder of Genesis Computers, has about 28+ years experience in software industry. He worked for OMC computers, Intergraph India Private Ltd., Intergraph Corporation (USA), and was a consultant to D.E. Shaw India Software Private Ltd and iLabs Limited . He has more than 25 years of teaching experience, and has conducted training for the corporates like ADP, APSRTC, ARM, B.H.E.L, B2B Software Technologies, Cambridge Technology Enterprises Private Limited, CellExchange India Private Limited, Citicorp Overseas Software Limited, CMC Centre (Gachibowli, Posnett Bhavan), CommVault Systems (India) Private Limited, Convergys Information Management (India) Private Limited, D.E. Shaw India Software Private Limited, D.R.D.L, Deloitee Consulting India Private Limited, ELICO Limited, eSymbiosis ITES India Private Limited, Everypath Private Limited, Gold Stone Software, HCL Consulting (Chennai), iLabs Limited, Infotech Enterprises, Intelligroup Asia Private Limited, Intergraph India Private Limited, Invensys Development Centre India Private Limited, Ivy Comptech, JP Systems (India) Limited, Juno Online Services Development Private Limited, Malpani Soft Private Limited, Mars Telecom Systems Private Limited, Mentor Graphics India Private Limited, Motorola India Electronics Limited, NCR Corporation India Private Limited, Netrovert Software Private Limited, Nokia India Private Limited, Optima Software Services, Oracle India Private Limited, Polaris Software Lab Limited, Qualcomm India Private Limited (Chennai, Hyderabad, Bangalore), Qualcomm China, Quantum Softech Limited, R.C.I, Renaissance Infotech, Satyam Computers, Satyam GE, Satyam Learning Centre, SIS Software (India) Private Limited, Sriven Computers, Teradata - NCR, Tanla Solutions Limited, Timmins Training Consulting Sdn. Bhd, Kuala Lumpur, Vazir Sultan Tobacco, Verizon, Virtusa India Private Limited, Wings Business Systems Private Limited, Wipro Systems, Xilinx India Technology Services Private Limited, Xilinx Ireland, Xilinx Inc (San Jose).

Genesis InSoft Limited
1-7-1072/A, RTC * Roads, Hyderabad - 500 020

rtapadia@genesisinsoft.com
www.genesisinsoft.com

Software design is hard; we need all the help we can get.
Bjarne Stroustrup, 1991

1.1 Introduction

Object Oriented Programming (OOP) is a new way of approaching the job of programming. Programming techniques have drastically changed since the invention of computers to accommodate the increasing complexity of programs. Earlier punched cards were used; later assembly language & finally the high-level languages were introduced that gave the programmer more tools with which to handle complexity (Fortran). The 1960s gave birth to structured programming languages like C & Pascal. With the structured programming languages it was possible to write moderately complex problems easily but once the project reaches a certain size its complexity becomes too difficult for a programmer to manage. To solve this problem OOP was invented.

OOP is a method of implementation in which programs are organized as cooperative collection of objects, each of which represents an instance of some class, and whose classes are all members of a hierarchy of classes united via inheritance relationships - Grady Booch.

Object oriented design is a method of design encompassing the process of object oriented decomposition and a notation for depicting both logical and physical as well as static and dynamic models of the system under design - Grady Booch.

OOP takes the ideas of structured programming & combines them with powerful, new concepts that encourage you to look at the task of programming in a new light. OOP allows you to easily decompose a problem into subgroups of related parts. Then, you translate these subgroups into self-contained units called objects. OOP have three things in common - objects, polymorphism & inheritance.

The most important feature of an object Oriented language is the object. An object is a logical entity containing data & code that manipulates the data. With an object, some of the code or data may be private to the object & inaccessible by anything outside the object. In this way, an object provides a significant level of protection against accidental modification or incorrect use. The linkage of code & data in this way is often referred to as data encapsulation.

Polymorphism allows one name to be used for several related but slightly different purposes. The purpose of polymorphism is to let one name be used to specify a general class of action. Depending upon what type of data it is dealing with, a specific instance of a general class is executed. For example, you might have a program that defines three different types of stacks - one for integer values, one for floating point values & one for longs. If you create three sets of functions called push() & pop(), the compiler will select the correct one depending on the type of data with which it is called.

Inheritance

Inheritance is the process by which one object can acquire the properties of another object. This concept supports the concept of classification. For example, an apple is part of fruit class that is under the larger food class. Without the use of classification each object would have to define all its characteristics explicitly. Using classification, an object need only define those qualities that make it unique within the class.

The C++ programming language was developed at AT&T Bell Laboratories in the early 1980s by Bjarne Stroustrup. It is an evolution of C language, which extends in three important ways.

1. It provides support for creating and using data abstractions.
2. It provides support for object-oriented design and programming.
3. It provides various nice improvements over existing C constructs.

1.2 Software Quality Factors

The primary aim of software engineering is to help produce quality software. We are going to look at some techniques that have a dramatic potential for improving the quality of software products. We want our programs to be fast, reliable, easy to use, readable, modular, structured and so on. We have two issues to consider here, external factors (speed, ease of use etc.) and internal factors (modular or readable etc.). To an end user the external factors matter as he doesn't care whether the program is readable or not; if the computations take longer to compute. However, the internal factors are the ones, which ensure that the external factors are satisfied.

1.2.1 External Quality Factors

Correctness - It is the ability of software products to exactly perform their tasks, as defined by the requirements and specifications. It is one of the most important quality. If a system does not do what it is supposed to do then everything else about it is of less importance.

Robustness - It is the ability of the software systems to function even in abnormal conditions. This is to take care of situations not generally specified in the specs. If a catastrophic situation occurs, the program should terminate its execution cleanly or exit gracefully, Fig. 1.1 shows the difference between correctness and robustness.

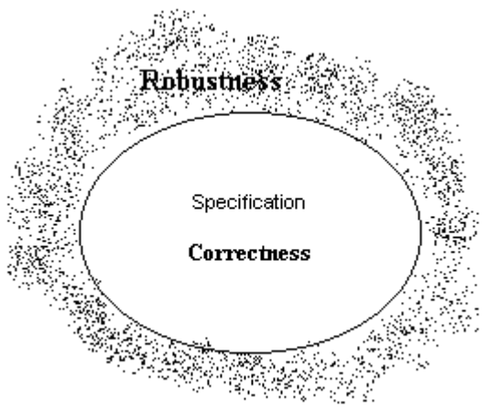

Fig. 1.1 Robustness vs. Correctness

Extendibility - It is the ease with which software products may be adapted to changes to specification. As programs grow bigger, they become harder to adapt. It often seems that a large system is a giant fragile construction in which pulling out any one brick will cause the whole edifice to collapse. The principles that improve extendibility are:

- Design simplicity - a simple architecture will always be easier to adapt to changes than a complex one.

- Decentralize - the more autonomous the modules in a software architecture, the more the likelihood that a simple change will affect just one module, or a small number of modules, rather than trigger off a chain reaction of changes over the whole system.

Reusability - It is the ability of software products to be reused, in whole or in part, for new applications. This requirement comes from the fact that many software systems follow common patterns; it should be possible to exploit this commonality and avoid reinventing solutions to problems that have been encountered before.

Compatibility - It is the ease with which software products may be combined with others. Since software products are not developed in vacuum, they need to interact with each other. The key to compatibility lies in homogeneity of design, and in agreeing on standardized conventions for inter-program communication, for example:

- Standardized file formats, as in Unix system, where every text file is simply a sequence of characters.

The other aspects of software quality are

1. Efficiency - It is the good use of hardware resources, such as processors, memory etc.
2. Portability - It is the ease with which software products may be transferred to various hardware and software environment.
3. Verifiability - It is the ease of preparing acceptance procedures, particularly test data and procedures for detecting failures etc.
4. Integrity - It is the ability of software systems to protect their various components (programs, data and documents) against unauthorized access and modifications.
5. Ease of use - It is the ease of learning how to use software systems, operating them, preparing input data, interpreting results, recovering from errors etc.

Fig. 1.2 shows the breakdown of maintenance costs.

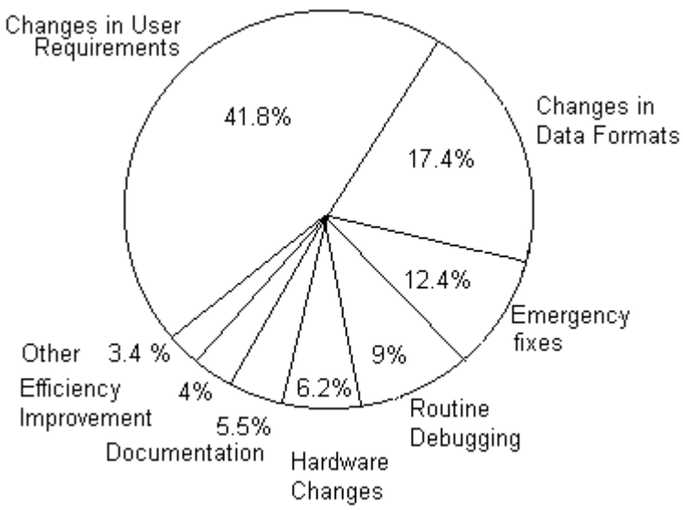

Fig. 1.2 Breakdown of maintenance costs

Most of the people consider software as only the software development phase. However, it is estimated that about 70% of the cost of software is devoted to maintenance. Software maintenance doesn't mean wear and tear because of repeated usage (like car or house maintenance); instead it refers to modifications in specifications, enhancements or bug fixing. According to Fig. 1.2 more than two-fifths of maintenance activities are because of extensions and modifications requested by the user. The second major factor is changes in data format.

1.2.2 Modularity

We need to have flexible system architecture in order to achieve the goals of software extensibility, reusability and compatibility. Modular programming refers to building an application consisting of small programs. If the modules themselves are not self-sustaining, coherent and organized in robust architectures it doesn't really benefit in the long run. The criteria that evaluate design methods with respect to modularity are

1. Modular decomposability - It is a design methodology which decomposes a problem into a set of several subproblems. The top-down design method helps designers to start with a most abstract description of the system's function, each subsystem being decomposed at each step into a small number of simpler ones, until all elements obtained are of a sufficiently low level of abstraction as to allow for direct implementation (tree structure). Fig. 1.3 illustrates decomposability.

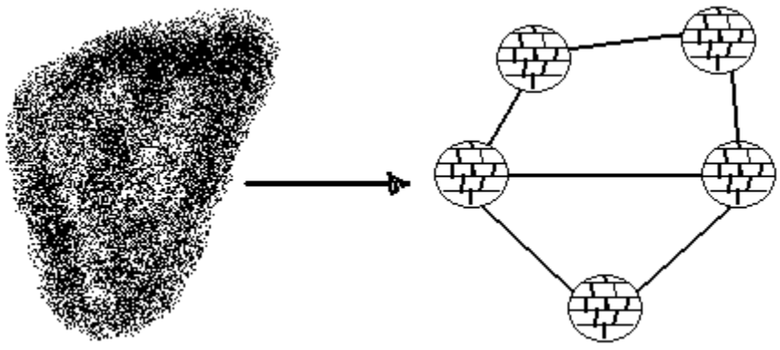

Fig. 1.3 Decomposability

2. Modular composability - It is a mechanism by which software systems may be combined to produce new systems. Composability is directly combined with the reusability problems. Its aim is to find ways to design pieces of software performing well-defined tasks and usable in widely different contexts. Fig. 1.4 illustrates composability.

3. Modular understandability - It refers to the understanding of the modules by the non-developer of the modules by looking at a few neighboring modules if required. This becomes very critical when maintaining a software system, as the maintenance problems require the person to understand the modules before any fixes can be made. Fig. 1.5 illustrates understandability.

4. Modular continuity - If a small change in the specifications results in a change of just one module, or few modules then the system satisfies modular continuity. It means that small changes should affect individual modules in the structure of the system rather than the structure itself. Fig. 1.6 illustrates continuity.

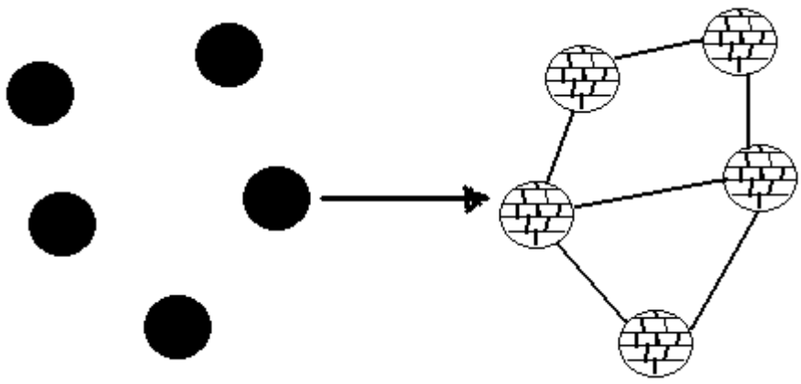

Fig. 1.4 Composability

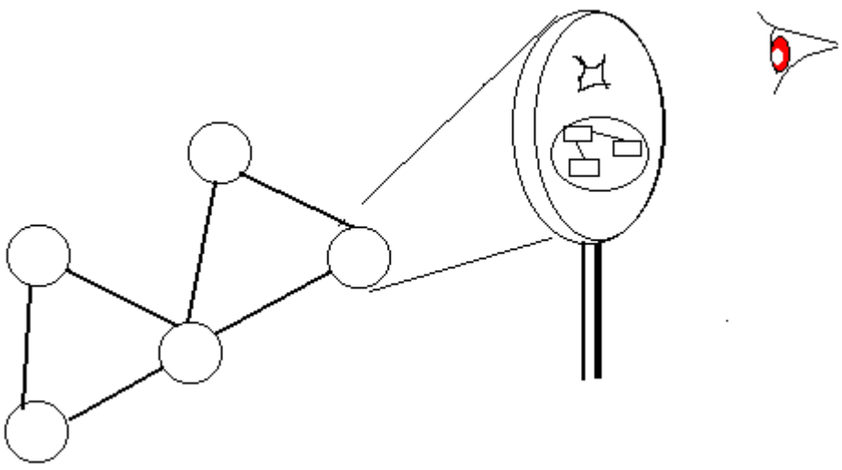

Fig. 1.5 Understandability

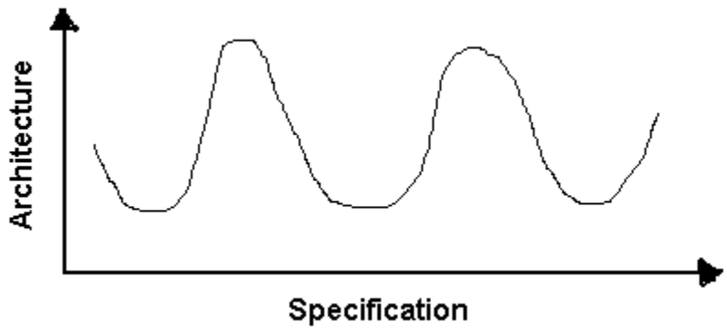

Fig. 1.6 Continuity

5. Modular protection - If a method yields architecture in which the effect of abnormal condition occurring at run-time in a module will remain confined to that module, or at least will propagate to a few neighboring modules then it is said to be modular protected. It doesn't refer to avoidance of errors but to the propagation of errors. Fig. 1.7 illustrates protection.

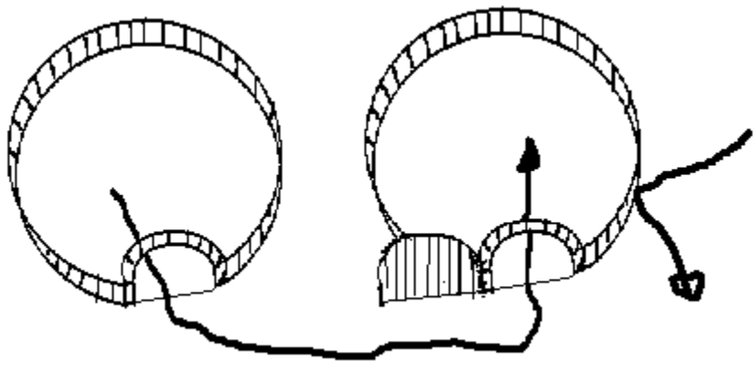

Fig. 1.7 Protection

The principles which must be observed to ensure proper modularity are

Linguistic Modular Units - Modules must correspond to syntactic units in the language used. The term language refers to a programming language, a program design language, a specification language etc. In a programming language each module should be separately compilable.

Few Interfaces - Every module should communicate with as few others as possible. Modules may interact with each other; share data structures etc., however such interactions should be limited. Fig. 1.8 illustrates the different types of module interconnection structure (one-many, one-one, many-many). If there are too many relations between modules, then the effect of a change or of an error may propagate to a large number of modules.

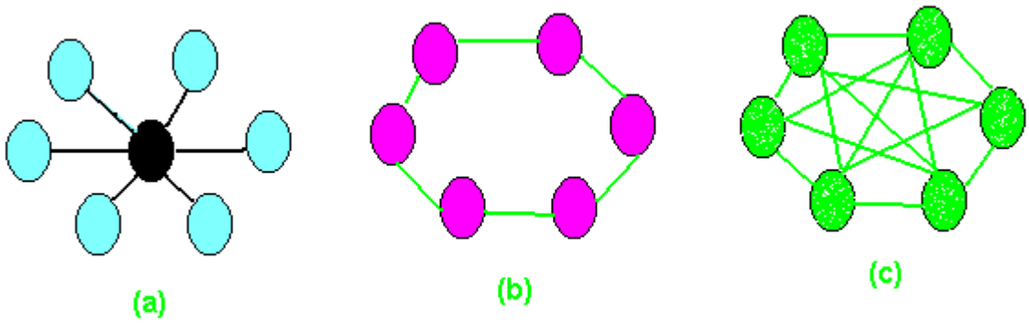

Fig. 1.8 Module Interconnection structures

Small Interface - If two modules communicate at all they should exchange as little information as possible. It refers to the size of intermodule connection rather than to their number. Fig. 1.9 illustrates the above concept. Fig. 1.9 illustrates Intermodule communication.

Explicit interfaces - Whenever modules A and B communicate this must be obvious from the text of A or B or both. The concept of decomposability, composability, continuity and understandability are greatly affected if there is no proper documentation of what the modules are doing and how they get affected when a particular parameter is changed. Fig. 1.10 illustrates data sharing.

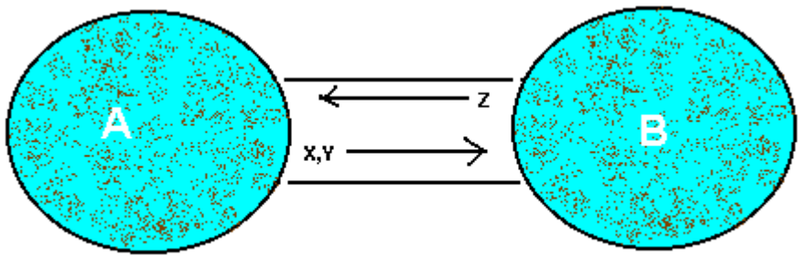

Fig. 1.9 Intermodule Communication

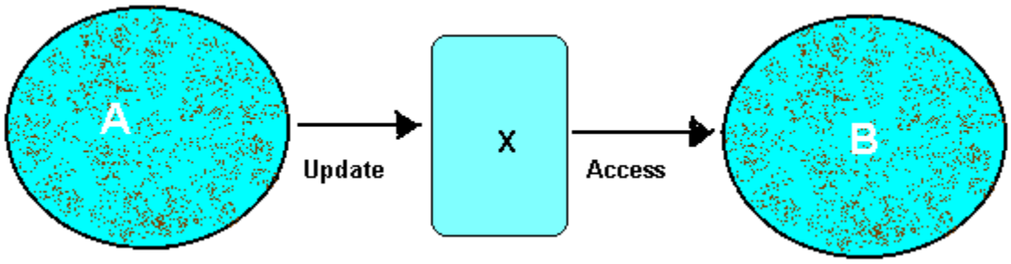

Fig. 1.10 Data Sharing

Information Hiding - All information about a module should be private to the module unless it is specifically declared public. The fundamental idea behind this principle is the continuity concept. If a module changes, but only in a way that affects its private elements, not the interface then other modules that use it, called client modules, will not be affected. For example we create a circle using the center point, radius. If the center point changes from the cartesian coordinates to the polar coordinates then the person who is using the circle function should not be affected (Public interface should not change just because a private data has changed). Information hiding does not imply protection in the sense of security restrictions (Unix concept of owner, group and others). Fig. 1.11 illustrates information hiding.

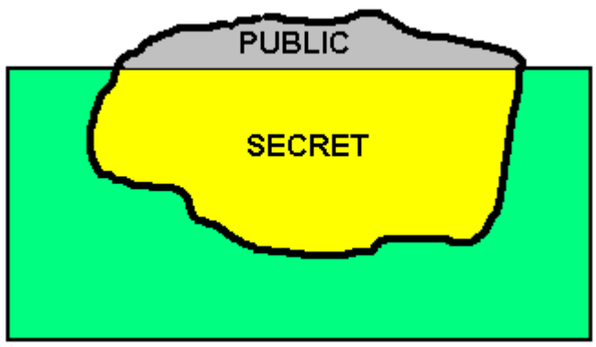

Fig. 1.11 Information Hiding

1.2.3 Reusability

Most of us who have written or browsed through software code have found that they all involve some common functionality like sorting, searching, reading, writing, comparing, traversing, allocating etc. This common set of functionality has been written multiple times without actually

using the ones that are readily available. This boils down to the fundamental question of why we should be reinventing the wheel every time. We would be better of using the things already available, as it would take less time to develop which means reduction in cost (both development and verification).

Reusability has really not caught on because of many factors. One of the factors is the economics. If a very general and reusable code is given to a customer he's probably not going to request for any other software system from you. Another factor that limits reusability is the awareness of similar software in the first place. To incorporate this, databases should exist which informs the software developer of the things already available.

1.2.3.1 Approaches to reusability

Reusability has been incorporated to some extent. The different approaches to reusability are:

- Source code Reusability - Using the source code for carrying out the application development (using Unix source code that is readily available).
- Reusability of Personnel - Using the previous experience of software engineer to work on a similar project.
- Reusability of Design - Using the old design information (blueprints consisting of design structures) to carry out new designs.

1.2.3.2 Requirements on Module Structures

Let's look at a simple example of table searching. Element say x, is given, as well as a set of similar elements, and the program should determine whether or not x appears in t. The general code is going to look the same each time. Start at some position in the table t, then start exploring the table from that position, each time checking whether the element found at the current position is the one we want, and, if not, move to another position. This process terminates when either the element has been found or not. Fig 1.12 illustrates the table-searching algorithm (Eiffel programming language)

```
search(x : ELEMENT, t : TABLE_OF_ELEMENT)
return boolean is pos : POSITION
begin
        pos : =   INITIAL_POSITION(x, t);
        while not EXHAUSTED (pos, t) and then not found (pos, x, t)
        do
                pos := NEXT (pos, x, t):
        end;
        return not EXHAUSTED(pos, t)
end
```

Fig. 1.12 Table searching algorithm

Now the question boils down to, how do we find module structures that will yield appropriate reusable components? We are going to look at some of the issues that need to be solved before we can hope to produce reusable code (we will look at the template for a general searching algorithm).

Variation in types: The searching algorithm should be applicable to different instances of the type ELEMENT (it should be able to search for an integer in a table of integers, an employee record in the corresponding table etc.).

Variation in data structures and algorithms: We should be able to write a general-purpose module, which should adapt to many different data structures and associated search algorithm (sequential tables, arrays, files etc.).

Related routines: It is not enough to know only about the table-searching algorithm, it must also be accompanied by table insertion, table deletion and the other related functions.

Representation independence: flexible modular systems should enable clients to specify an operation without knowing how it is implemented. For example the client should be able to make a call like

Present := Search(x, t);

Without knowing what kind of table is at the time of the call. If various searching algorithms are provided the underlying mechanism should be able to find the appropriate ones without client intervention. This criteria is not simple reusability but extensibility (i.e. t may change at run time and because of which a new searching algorithm must be invoked). This simple check for all the different options can't be done using if-else or switch statements, as that would mean knowing all the searching implementations routines. What would be more desirable is at run time it should decide on the type of search function to call based on the table of elements (dynamic binding).

Commonality within subgroups: Whenever we want to have a reusable code we should find the commonality that may exist within a set of related functions. In the table searching mechanism, a typical example is the subgroup comprising of all sequential tables - it may be implemented as sequential arrays, linked lists or sequential files. The algorithm is nearly the same for all sequential implementation; the differences affect a small set of primitive operations used by the algorithm- starting at first position, moving to the next, testing for final positions. Fig 1.13 lists the code for sequential table searching. Fig 1.14 lists the implementation variants for primitive operation

```
search(x : ELEMENTS, t : SEQUENTIAL_TABLE)
boolean is pos: POSITION
begin
        START_SEARCH;
        while not EXHAUSTED and then not found (pos,  x, t)
        do
                MOVE_TO_NEXT
        end;
        return not EXHAUSTED
end
```

Fig. 1.13 Sequential table searching algorithm

	Sequential Array	Linked List	Sequential File
START_SEARCH	j := 1	k := head	Rewind
MOVE_TO_NEXT	j := j + 1	k := k.next	Readnext
EXHAUSTED	j > size	k = null	End of file

Fig. 1.14 Implementation variants for primitive operations

1.2.3.3 Methods of Reusability

- Routines - A routine is a procedure, function, subroutine, subprogram etc. We need to build libraries of routines that can then be used by multiple applications without each application having to duplicate the same. Decomposition of software systems into routines is what is generally done using the top-down approach. A drawback of this approach is that it does not have information of the related routines.
- Packages - A package refers to a group of related routines combined together. It may contain data structures and associated operations. The advantage is that all program entities that relate to an important conceptual part of the system are at the same place, compiled together.

1.2.4 Overloading and Genericity

Overloading is the ability to attach more than one meaning to a name appearing in a program. This is useful for client programmers. For example, if we write several table implementations, each defined by a separate type declaration, overloading allows us to give the same name, say search, to all the associated search procedures. Genericity is the ability to define parameterized modules. This is useful for module developers. Such a module called a generic module is not directly usable. In most cases the parameters (formal generic parameters) stand for types. Fig 1.15 illustrates the genericity concept.

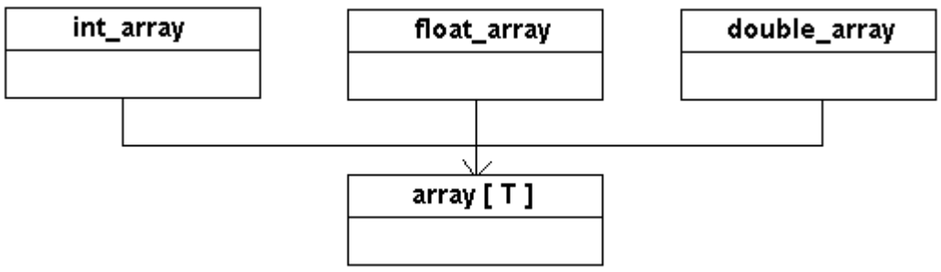

Fig. 1.15 Genericity example

In this example array [T] is a generic class, where T is a formal generic parameter. The actual classes array [integer], array[float] and array[double] are instances of array[T].

1.3 The Road to Object Orientedness

A software system is a set of mechanisms for performing certain actions on certain data. A software designer is always confronted with the fundamental choice of selecting the structure based on either the data or actions. Any successful system inevitably undergoes numerous changes over its lifetime. To evaluate the quality of any architecture, we should not just consider how easily this architecture is initially obtained. It is just as important, and perhaps more, to ascertain how well the architecture will withstand the process of change. For example, a payroll program that initially used to produce paychecks from time cards will, after a while, be extended to gather statistics, produce tax information, maintain repository of employee data etc., it's initial functions will be modified.

1.3.1 Top Down Design Method

Classical methods typically use the functions, not the objects, as the basis. The top-down design is based on the idea that software should be based on stepwise refinement of the system's abstract function. The process of top-down refinement may be described as the development of a tree. Nodes of the tree represent elements of the decomposition; branches show the relation. Fig 1.16 shows the top-down tree hierarchy structure.

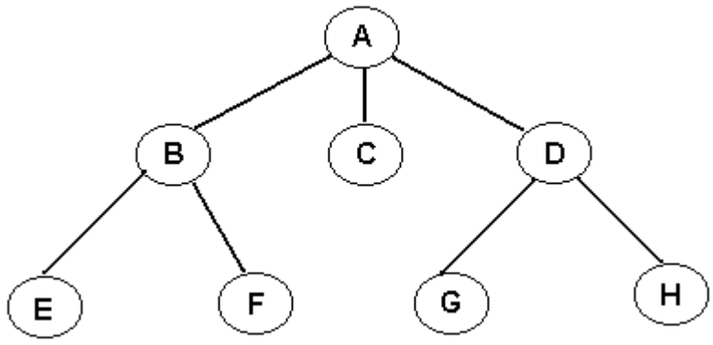

Fig. 1.16 The Tree hierarchy

The advantages of top-down design are:

- Logical
- Well organized
- Encourages orderly development

The flaws of top-down design are:

- Fails to take into account the evolutionary nature of software systems
- The very notion of a system being characterized by one function is questionable
- Using the function as basis often means that data structure aspect is neglected
- Does not promote Reusability

Lets take an example of an operating system. It is viewed as a system that offers a number of services - allocating CPU time, managing memory, handling input-output devices, interpreting user commands etc. A well-organized operating system will be more or less built in modules centered on these groups of functions. Top-down functional decomposition doesn't yield such architecture. It forces us to ask the question "what is the topmost function". Top-down method is poorly adapted to the development of significant software systems. It is a useful paradigm for small programs and individual algorithms, but it doesn't scale up to sizeable practical systems.

1.3.2 Object oriented Design (OOD)

OOD is the method that leads to software architectures based on the objects every system or subsystem manipulates (rather than "the" function it is meant to ensure). OOD motto can be expressed as

> Ask not first what the system does:
> Ask WHAT it does it to!

Using OOD, the designer will stay away, as long as possible, from the need to describe and implement the topmost function of the system. Instead they will analyze the classes of objects of the system. The simple idea of looking at the data first and forgetting the immediate purpose of the system is the key to reusability and extendibility. The above definition provides a general guideline for designing software but it raises a number of issues:

How to find the objects?
How to describe the objects?
How to describe the relations and commonalties between objects?
How to use objects to structure programs?

1.3.3 Steps Towards Object-Oriented Happiness

1. Object-based modular structure - Systems are modularized on the basis of their data structures
2. Data abstraction - Objects should be described as implementation of abstract data types.
3. Automatic memory management - Unused objects should be deallocated by the underlining language system, without programmer intervention.
4. Classes - Every non-simple type is a module, and every high level module is a type.
5. Inheritance - A class may be defined as an extension or restriction of another.
6. Polymorphism & Dynamic binding - Program entities should be permitted to refer to objects of more than one class, and operations should be permitted to have different realizations in different classes.
7. Multiple & Repeated Inheritance - It should be possible to declare a class as heir to more than one class and more than once to the same class. Refer to Fig. 1.17.

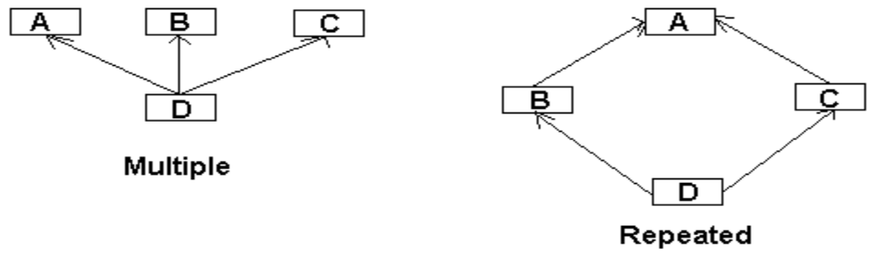

Fig 1.17 Multiple and repeated inheritance

1.4 Exercise

1. What is modularity? What are the principles of modularity?
2. What are the aspects of software quality?
3. What are the requirements on module structures?
4. What is the difference between overloading and genericity?
5. What is the difference between procedural and object oriented programming?
6. Briefly explain the concept of reusability with respect to software programming?
7. What does object oriented design mean? What are the steps towards object oriented happiness?
8. Why do you use an OO language? Give three reasons?
9. How are data and functions organized in an object-oriented program?
10. Describe inheritance and Polymorphism as applied to OOP?