UNIT - I

Syllabus: The Object Model-The Evolution of the Object Model: The generations of programming languages, the topology of Programming languages. Foundations of the Object Model: Object Oriented Analysis, Object Oriented design, Object Oriented Programming. Elements of the Object Model: Programming Paradigm(programming style), The Major and Minor Elements of the Object Models, Abstraction, Encapsulation, Modularity, Hierarchy(single inheritance, multiple inheritance, Aggregation), Static and Dynamic Typing, Concurrency, Persistence.

The object model:

Object-oriented technology is built on a sound engineering foundation, whose elements we collectively call the object model of development or simply the object model. The object model encompasses the principles of abstraction, encapsulation, modularity, hierarchy, typing, concurrency, and persistence.

The evaluation of the Object Model:

In this section we will examine the evolution of the tools of our profession to help us understand the foundation and emergence of objectoriented technology.

The Generations of Programming Languages

Wegner has classified some of the more popular high-order programming languages in generations arranged according to the language features they first introduced.

- **First-generation languages (1954–1958):**

FORTRAN I	Mathematical expressions
ALGOL 58	Mathematical expressions
Flowmatic	Mathematical expressions
IPL V	Mathematical expressions

■ **Second-generation languages (1959–1961)**

FORTRAN II	Subroutines, separate compilation
ALGOL 60	Block structure, data types
COBOL	Data description, file handling
Lisp	List processing, pointers, garbage collection

■ **Third-generation languages (1962–1970)**

PL/1	FORTRAN + ALGOL + COBOL
ALGOL 68	Rigorous successor to ALGOL 60
Pascal	Simple successor to ALGOL 60
Simula	Classes, data abstraction

■ **The generation gap (1970–1980)**

C	Efficient; small executables
FORTRAN 77	ANSI standardizatio

■ **Object-orientation boom (1980–1990)**

Smalltalk 80	Pure object-oriented language
C++	Derived from C and Simula
Ada83	Strong typing; heavy Pascal influence
Eiffel	Derived from Ada and Simula

■ **Emergence of frameworks (1990–today)**

Visual Basic	Eased development of the graphical user interface (GUI) for Windows applications
Java	Successor to Oak; designed for portability
Python	Object-oriented scripting language
J2EE	Java-based framework for enterprise computing

.NET	Microsoft's object-based framework
Visual C#	Java competitor for the Microsoft .NET Framework
Visual Basic .NET	Visual Basic for the Microsoft .NET Framework

Topologies of Programming languages:

The Topology of First- and Early SecondGeneration Programming Languages

Let's consider the structure of each generation of programming languages. In below Figure, we see the topology of most first- and early second-generation programming languages. By topology, we mean the basic physical building blocks of the language and how those parts can be connected. In this figure, we see that for languages such as FORTRAN and COBOL, the basic physical building block of all applications is the subprogram (or the paragraph, for those who speak COBOL).

- Applications written in these languages exhibit a relatively flat physical structure, consisting only of global data and subprograms.
- The arrows in this figure indicate dependencies of the subprograms on various data.
- During design, one can logically separate different kinds of data from one another, but there is little in these languages that can enforce these design decisions.
- An error in one part of a program can have a devastating ripple effect across the rest of the system because the global data structures are exposed for all subprograms to see.

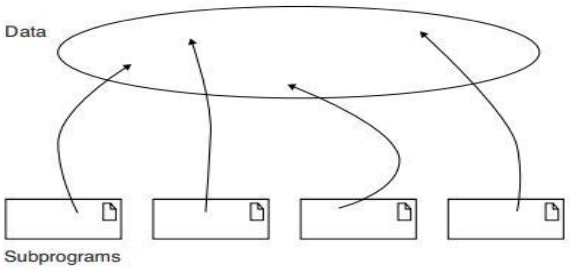

Figure The Topology of First- and Early Second-Generation Programming Languages

The Topology of Late Second- and Early Third-Generation Programming Languages:

By the mid-1960s, programs were finally being recognized as important intermediate points between the problem and the computer. "The first software abstraction, now called the 'procedural' abstraction, grew directly out of this pragmatic view of software.

The realization that subprograms could serve as an abstraction mechanism had three important consequences.

- **First**, languages were invented that supported a variety of parameter-passing mechanisms.
- **Second**, the foundations of structured programming were laid, manifesting themselves in language support for the nesting of subprograms and the development of theories regarding control structures and the scope and visibility of declarations.
- **Third**, structured design methods emerged, offering guidance to designers trying to build large systems using subprograms as basic physical building blocks.
- Thus, it is not surprising, as Figure shows below, that the topology of late second- and early third-generation languages is largely a variation on the theme of earlier generations.
- It is still fails to address the problems of programming-in-the-large and data design.

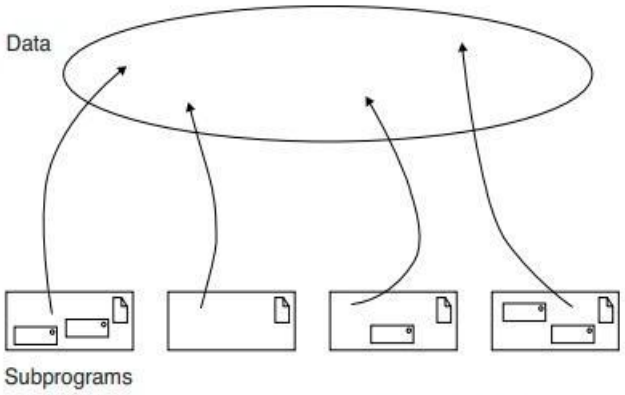

Figure The Topology of Late Second- and Early Third-Generation Programming Languages

The Topology of Late Third-Generation Programming Languages:

Starting with FORTRAN II, and appearing in most late third-generation program languages, another important structuring mechanism evolved to address the growing issues of programming-in-the-large. Larger programming projects meant larger development teams, and thus the need to develop different parts of the same program independently. The answer to this need was the separately compiled module, which in its early conception was little more than an arbitrary container for data and subprograms, as Figure below shows.

Most languages of this generation, while supporting some sort of modular structure, had few rules that required semantic consistency among module interfaces.

- A developer writing a subprogram for one module might assume that it would be called with three different parameters: a floating-point number, an array of ten elements, and an integer representing a Boolean flag.
- In another module, a call to this subprogram might incorrectly use actual parameters that violated these assumptions: an integer, an array of five elements, and a negative number.
- Similarly, one module might use a block of common data that it assumed as its own, and another module might violate these assumptions by directly manipulating this data.
- Most of these languages address the errors while program execution.

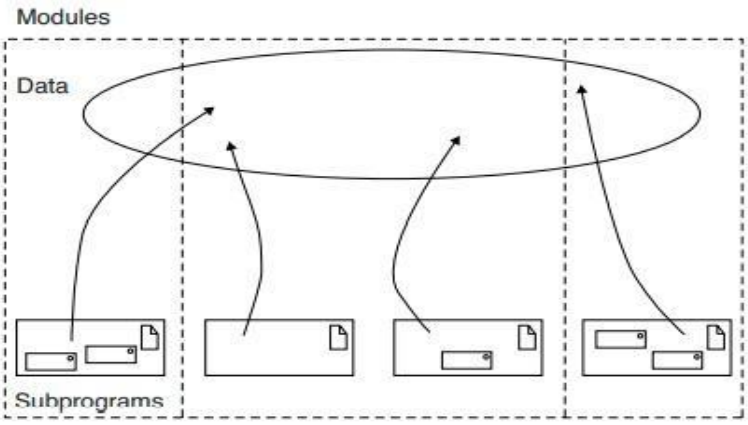

Figure The Topology of Late Third-Generation Programming Languages

The Topology of Object-Based and Object-Oriented Programming Languages:

Smalltalk, Object Pascal, C++, Ada, Eiffel, and Java, these languages are called object-based or object-oriented. Figure illustrates the topology of such languages for small to moderate-sized applications.

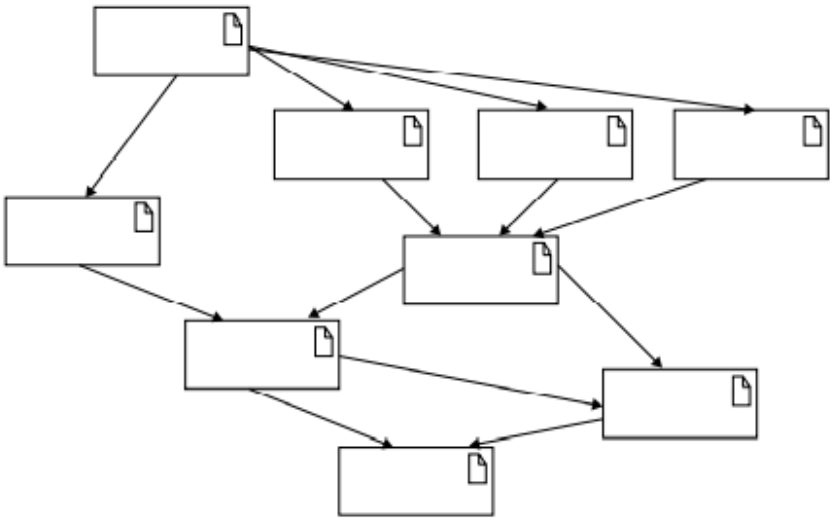

Figure 2–4 The Topology of Small to Moderate-Sized Applications Using Object-Based and Object-Oriented Programming Languages

The physical building block in such languages is the module, which represents a logical collection of classes and objects instead of subprograms, as in earlier languages.

To state it another way, "If procedures and functions are verbs and pieces of data are nouns, a procedure-oriented program is organized around verbs while an object-oriented program is organized around nouns".

For this reason, the physical structure of a small to moderate-sized object-oriented application appears as a graph, not as a tree, which is typical of algorithmically oriented languages.

In large systems, we find clusters of abstractions built in layers on top of one another. At any given level of abstraction, we find meaningful collections of objects that collaborate to achieve some higher-level behavior. If we look inside any given cluster to view its implementation, we unveil yet another set of cooperative abstractions. This topology is shown in Figure

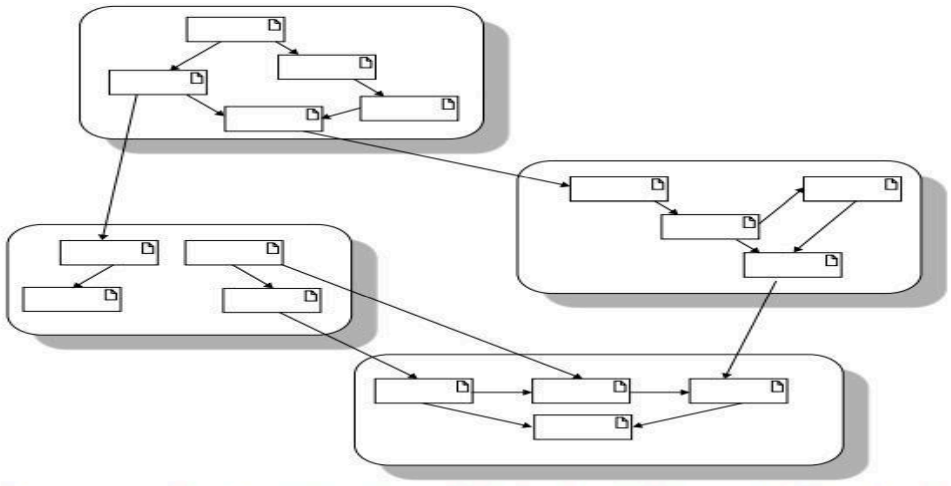

Figure The Topology of Large Applications Using Object-Based and Object-Oriented Programming Languages

FOUNDATIONS OF OBJECT MODEL

Structured design methods evolved to guide developers who were trying to build complex systems using algorithms as their fundamental building blocks. Similarly,object-oriented design methods have evolved to help developers exploit the expressive power of object-based and object-oriented programming languages, using the class and object as basic building blocks.

Object-Oriented Programming (OOP)

What is object-oriented programming (OOP)?

We define it as follows: Object-oriented programming is a method of implementation in which programs are organized as cooperative collections 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.

There are three important parts to this definition: (1) Object-oriented programming uses objects, not algorithms, as its fundamental logical building blocks (2) each object is an instance of some class; and (3) classes may be related to one another via inheritance relationships (the "is a" hierarchy). A program may appear to be object-oriented, but if any of these elements is missing, it is not an object-oriented program.

Language is object-oriented if and only if it satisfies the following requirements:

■ It supports objects that are data abstractions with an interface of named operations and a hidden local state.

■ Objects have an associated type [class].

■ Types [classes] may inherit attributes from supertypes [superclasses].

Object-Oriented Design

What is objectoriented design (OOD)? We suggest the following:

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

There are two important parts to this definition: object-oriented design (1) leads to an object-oriented decomposition and (2) uses different notations to express different models of the logical (class and object structure) and physical (module and process architecture) design of a system, in addition to the static and dynamic aspects of the system.

The support for object-oriented decomposition is what makes object-oriented design quite different from structured design: The former uses class and object abstractions to logically structure systems, and the latter uses algorithmic abstractions. We will use the term object-oriented design to refer to any method that leads to an object-oriented decomposition.

Object-Oriented Analysis (OOA)

Object-oriented analysis (OOA) emphasizes the building of real-world models, using an object-oriented view of the world:

"Object-oriented analysis is a method of analysis that examines requirements from the perspective of the classes and objects found in the vocabulary of the problem domain."

How are OOA, OOD, and OOP related? Basically, the products of object-oriented analysis serve as the models from which we may start an object-oriented design; the products of object-oriented design can then be

used as blueprints for completely implementing a system using object-oriented programming methods.

Elements of Object Model:

Bobrow and Stefik define a programming style as "a way of organizing programs on the basis of some conceptual model of programming and an appropriate language to make programs written in the style clear" . They further suggest that there are five main kinds of programming styles, listed here with the kinds of abstractions they employ:

1. Procedure-oriented	Algorithms
2. Object-oriented	Classes and objects
3. Logic-oriented	Goals, often expressed in a predicate calculus
4. Rule-oriented	If–then rules
5. Constraint-oriented	Invariant relationships

There is no single programming style that is best for all kinds of applications. For example, rule-oriented programming would be best suited for the design of a knowledge base, and procedure-oriented programming would be best for the design of computation-intense operations. From our experience, the object-oriented style is best suited to the broadest set of applications; indeed, this programming paradigm often serves as the architectural framework in which we employ other paradigms.

Each of these styles of programming is based on its own conceptual framework. Each requires a different mindset, a different way of thinking about the problem. For all things object-oriented, the conceptual framework is the object model.

There are four major elements of this model:

1. Abstraction
2. Encapsulation
3. Modularity
4. Hierarchy

By major, we mean that a model without any one of these elements is not objectoriented.

The Meaning of Abstraction

We define an abstraction as follows:

"An abstraction denotes the essential characteristics of an object that distinguish it from all other kinds of objects and thus provide crisply defined conceptual boundaries, relative to the perspective of the viewer.**"**

An abstraction focuses on the outside view of an object and so serves to separate an object's essential behavior from its implementation.

"There is a spectrum of abstraction, from objects which closely model problem domain entities to objects which really have no reason for existence". From the most to the least useful, these kinds of abstractions include the following:

- **Entity abstraction:** An object that represents a useful model of a problem domain or solution domain entity

- **Action abstraction:** An object that provides a generalized set of operations, all of which perform the same kind of function

- **Virtual machine abstraction:** An object that groups operations that are all used by some superior level of control, or operations that all use some junior-level set of operations

- **Coincidental abstraction:** An object that packages a set of operations that have no relation to each other.

Example – When a class Student is designed, the attributes enrolment_number, name, course, and address are included while characteristics like pulse_rate and size_of_shoe are eliminated, since they are irrelevant in the perspective of the educational institution.

Abstraction: Student
Important char's: Roll_num, name, course, address
Responsibilities: 1210, mcs,ndl

Encapsulation

Encapsulation is the process of binding both attributes and methods together within a class. Through encapsulation, the internal details of a class can be hidden from outside. It permits the elements of the class to be accessed from outside only through the interface provided by the class.

Encapsulation tells us that we should know just an interface of class to be able to use it. We don't have to know anything about internals of class to be able to use it.

Let's check following real-world example. The best example of encapsulation could be a calculator. We understand its interface from first sight and we don't have to know how it works inside. We know that we can press 2+2 then = and see the result on display.

Modularity

Modularization consists of dividing a program into modules which can be compiled separately, but which have connections with other modules. We will use the definition of Parnas: 'The connections between modules are the assumptions which the modules make about each other

We may define modularity as follows:

"Modularity is the property of a system that has been decomposed into a set of cohesive and loosely coupled modules.**"**

Thus, the principles of abstraction, encapsulation, and modularity are synergistic. An object provides a crisp boundary around a single abstraction, and both encapsulation and modularity provide barriers around this abstraction.

Hierarchy

We define hierarchy as follows:

"Hierarchy is a ranking or ordering of abstractions."

The two most important hierarchies in a complex system:

1. Class structure (the "is a" hierarchy) and
2. Object structure (the "part of" hierarchy).

Examples of Hierarchy: Single Inheritance

Inheritance is the most important "is a" hierarchy, and as we noted earlier, it is an essential element of object-oriented systems. Basically, inheritance defines a relationship among classes, wherein one class shares the structure or behavior defined in one or more classes (denoting single inheritance and multiple inheritance, respectively). Inheritance thus represents a hierarchy of abstractions, in which a subclass inherits from one or more superclasses.

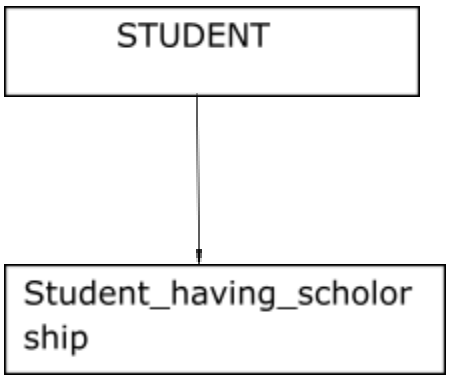

FIG: Single Inheritance

Examples of Hierarchy: Multiple Inheritance:

For example, suppose that we choose to define a class representing a kind of plant. Our analysis of the problem domain might suggest that flowering plants and fruits and vegetables have specialized properties that are relevant to our application. For example, given a flowering plant, its expected time to flower and time to seed might be important to us. Similarly, the time to harvest might be an important part of our abstraction of all fruits and vegetables.

For example, we can define a Rose class (see Figure1) that inherits from both Plant and FlowerMixin. Instances of the subclass Rose thus include the structure and behavior from the class Plant together with the structure and behavior from the class FlowerMixin.

Similarly, a Carrot class could be as shown in Figure 2. In both cases, we form the subclass by inheriting from two superclasses.

Now, suppose we want to declare a class for a plant such as the cherry tree that has both flowers and fruit. This would be conceptualized as shown in Figure 3

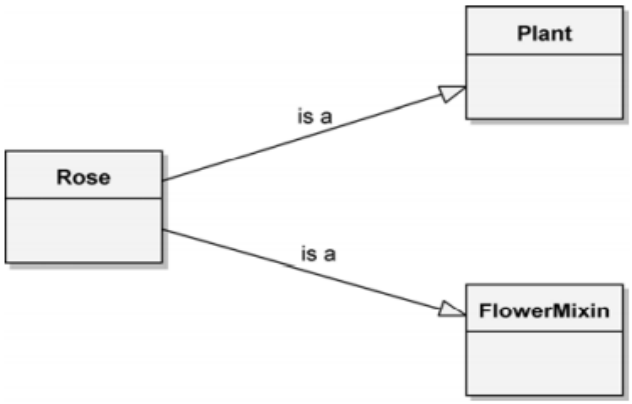

Fig1: The Rose Class, Which Inherits from Multiple Superclasses

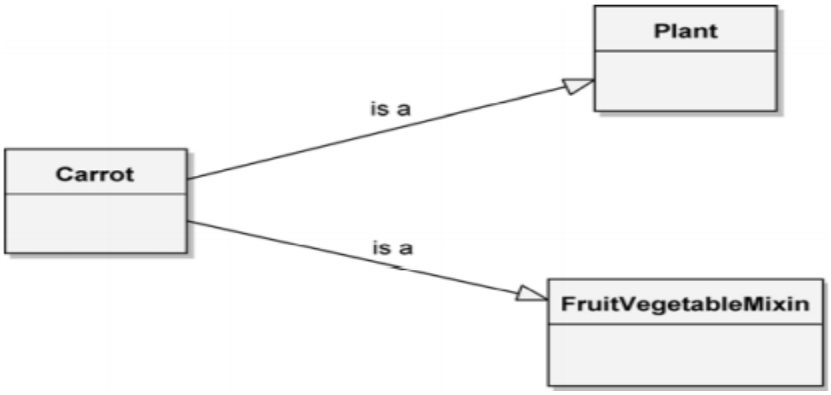

Fig 2: The Carrot Class, Which Inherits from Multiple Superclasses

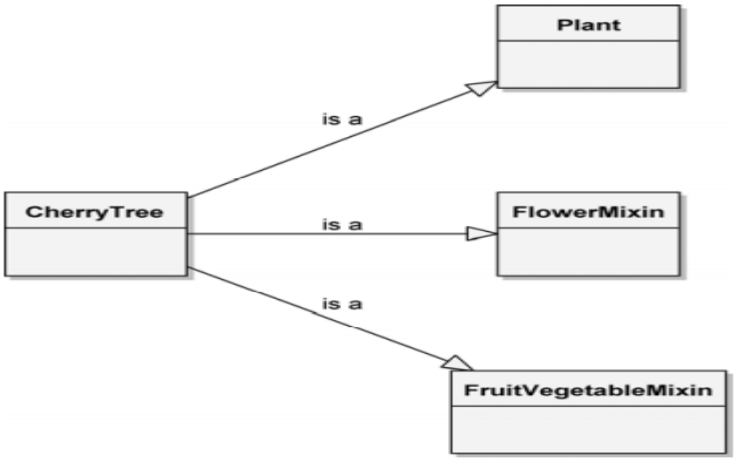

Fig 3: The CherryTree Class, Which Inherits from Multiple Superclasses

Generalization:

"Generalization and specialization represent a hierarchy of relationships between classes, where subclasses inherit from super-classes."

In the generalization process, the common characteristics of classes are combined to form a class in a higher level of hierarchy, i.e., subclasses are combined to form a generalized super-class. It represents an "is – a – kind – of" relationship. For example, "car is a kind of land vehicle", or "ship is a kind of water vehicle".

Examples of Hierarchy: Aggregation

Whereas these "is a" hierarchies denote generalization/specialization relationships, "part of" hierarchies describe aggregation relationships. For example, consider the abstraction of a garden. We can contend that a garden consists of a collection of plants together with a growing plan. In other words, plants are "part of" the garden, and the growing plan is "part of" the garden. This "part of" relationship is known as aggregation.

Minor elements:

There are three minor elements of the object model:

1. Typing
2. Concurrency
3. Persistence

By minor, we mean that each of these elements is a useful, but not essential, part of the object model.

Typing:

According to the theories of abstract data type, a type is a characterization of a set of elements.

Definition:

Typing is the enforcement of the class of an object, such that objects of different types may not be interchanged, or at the most, they may be interchanged only in very restricted ways.

The two types of typing are –

Strong Typing – Here, the operation on an object is checked at the time of compilation, as in the programming language Eiffel.

Weak Typing – Here, messages may be sent to any class. The operation is checked only at the time of execution, as in the programming language Smalltalk.

Concurrency

Concurrency in operating systems allows performing multiple tasks or processes simultaneously. When a single process exists in a system, it is said that there is a single thread of control. However, most systems have multiple threads, some active, some waiting for CPU, some suspended, and some terminated. Systems with multiple CPUs inherently permit concurrent threads of control; but systems running on a single CPU use appropriate algorithms to give equitable CPU time to the threads so as to enable concurrency.

In an object-oriented environment, there are active and inactive objects. The active objects have independent threads of control that can execute concurrently with threads of other objects. The active objects synchronize with one another as well as with purely sequential objects.

Persistence

An object occupies a memory space and exists for a particular period of time. In traditional programming, the lifespan of an object was typically the lifespan of the execution of the program that created it. In files or databases, the object lifespan is longer than the duration of the process creating the object. This property by which an object continues to exist even after its creator ceases to exist is known as persistence.

Advantages of Object Model:

1. The use of the object model helps us to exploit the expressive power of object-based and object-oriented programming languages
2. The use of the object model encourages the reuse not only of software but of entire designs, leading to the creation of reusable application frameworks

3. The use of the object model produces systems that are built on stable intermediate forms, which are more resilient to change.
4. It supports relatively hassle-free upgrades.
5. It reduces development risks, particularly in integration of complex systems.