SUBJECT: OBJECT ORIENTED ANALYSIS AND DESIGN USING UML

UNIT-1

(INTRODUCTION)

TOPICS:

- Introduction to OOAD
- The Structure of Complex systems.
- The Inherent Complexity of Software.
- Attributes of Complex System.
- Organized and Disorganized Complexity.
- Bringing Order to Chaos.
- Designing Complex Systems.
- Evolution of Object Model.
- Foundation of Object Model.
- Elements of Object Model.
- Applying the Object Model.

Object-oriented refers to a programming language, system or software methodology that is built on the concepts of logical objects. It works through the creation, utilization and manipulation of reusable objects to perform a specific task.

Object-oriented programming (OOP) refers to a type of computer programming (software design) in which programmers define not only the data type of a data structure, but also the types of operations (functions) that **can** be applied to the data structure.
Object–Oriented Analysis (OOA) is the process of identifying software engineering requirements and developing software specifications in terms of a software system's object model, which comprises of interacting objects.

object-oriented analysis phase of software development, the system requirements are determined, the classes are identified and the relationships among classes are identified.

The primary tasks in object-oriented analysis (OOA) are:

Identifying objects
Organizing the objects by creating object model diagram.
Defining the internals of the objects, or object attributes.
Defining the behavior of the objects, i.e., object actions Describing how the objects interact.
The common models used in OOA are use cases and object models.
Object-oriented design **(OOD)** is the process of using an *object-oriented* methodology to *design* a computing system or application. This technique enables the implementation of a

software solution based on the concepts of *objects*. OOD serves as part of the *object-oriented* programming (*OOP*) process or lifecycle.

Complexity

Systems: Systems are constructed by interconnecting components. The larger the number of these components and relationships between them, higher will be the complexity of the overall system.

complex Systems: A complex system refers to a system consisting of multiple interacting components in a hierarchy, where each of these components collectively perform certain task. some of the examples of complex systems are the structure of a personal computer, the structure of plants, the structure of animals, structure matter, The structure of Social institutions and the structure of human body etc.

A system that consist of one or more software subsystems which control a significant part of its overall behavior is called a software intensive system. some examples of complex software-intensive systems are rail systems, banking systems, health care systems and aviation systems so on.

Complexity: Complexity depends on the number of the components embedded in them as well as the relationships and the interactions between these components.

THE STRUCTURE OF COMPLEX SYSTEMS

Structure of complex systems is nothing but the interaction and behavior of components and their interconnectivity. it describes how the components carry out their tasks to achieve the goals. First study how complex systems in other disciplines are organized.

Examples of Complex Systems: The structure of personal computer, plants and animals, matter, social institutions are some examples of complex systems.

The structure of a Personal Computer: A personal computer is considered as the moderate complex system. Major elements are CPU, monitor, keyboard and some secondary storage devices. CPU consist of various parts such as primary memory, an ALU, and a bus to which peripheral devices are attached. Further every part is the combination of some other elements . for example An ALU is the combination of registers and control unit which are constructed from NAND gates, inverters and so on.

The structure of Plants: plants are considered as the complex multicellular organisms that shows complex behaviors such as photosynthesis and transpiration. these behaviors are due to the activities performed by different organ systems of plant. Plants consist of three major structures roots, stems, and leaves where each of these structures consists of some other specific structure and so on.

The structure of Animals: Animals exhibit a multicultural hierarchical structure in which collection of cells form tissues, tissues work together as organs, clusters of organs define systems (such as the digestive system) and so on.ewr

The structure of Matter:

The study of fields as diverse as astronomy and nuclear physics provides us with many other examples of incredibly complex systems. Spanning these two disciplines, we find yet another structural hierarchy.

The structure of Social institutions: In social institutions, group of people join together to accomplish tasks that can not be done by made of divisions which in turn contain branches which in turn encompass local offices and so on.

THE INHERENT COMPLEXITY OF SOFTWARE

Why Software is inherently Complex

The complexity of software is an essential property not an accidental one. The inherent complexity derives from four elements; the complexity of the problem domain, the difficultly of managing the developmental process, the flexibility possible through software and the problems of characterizing the behavior of discrete systems.

1. **The complexity of the problem domain:**
   - ➢ Problems that occur in software generally includes unavoidable complexity. where in numerable competing requirements are determined. for instance, assume the requirements of a multi engine aircraft, autonomous robot, cellular switching system. it is extremely difficult to understand the external functionality of these systems. thus the external complexity gives rise to arbitrary complexity.

2.The Difficulty of Managing the Development Process

- Management problems
- Need of simplicity

➢ The second reason is the complexity of the software development process. Complex software systems cannot be developed by single individuals. They require teams of developers. This adds extra overhead to the process since the developers have to communicate with each other about the intermediate artifacts they produce and make them interoperable with each other.

"The task of the software development team
is to engineer the illusion of simplicity."

3. The flexibility possible through software

- Software is flexible and expressive and thus encourages highly demanding requirements, which in turn lead to complex implementations which are difficult to assess.

4. The problem of characterizing the behavior of discrete systems

➢ Developing large software applications may consist of hundred or even thousands of variables. all these variables along with their current address, current values and calling stack represents the current state of application. As is implemented on digital computer, there are distinct states with in a single system.

ATTRIBUTES OF COMPLEX SYSTEM

The five Attributes of a complex system: There are five attributes common to all complex systems. They are as follows:

1. **Hierarchical and interacting subsystems**
   Every complex system is a composition of interrelated subsystems which inturn are the composition of some other systems and so on.
   this continues until a leaf level of components are attained which results in to hierarchic structure. this decomposable hierarchic structures are easy to understand and define.
   the architecture of a complex system can be developed by integrating the tasks performed by its components and the hierarchic relationship that exist between the components.
2. **Arbitrary determination of primitive components**
   The primitive components of every complex system are relatively inconsistent. the selection of the components are generally done by the observer of the system. class structure and the object structure are not completely independent each object in object structure represents a specific instance of some class.

3. **Stronger intra-component than inter-component link**

   Intra-component linkages are generally stronger than inter-component linkages.

   Due to this , the high frequency dynamics are separated from the low frequency dynamics. where high frequency dynamic contains internal structure of the components and low frequency dynamic contains interactions that take place between these components.

   form the differences existing between the intra components and inter components interaction, it can be inferred that there is a separation of concerns among the different components of a system.

4. **Combine and arrange common rearranging subsystems**

   Hierarchic systems are usually composed of only a few different kinds of subsystems in various combinations and arrangements. In other words, complex systems have common patterns. These patterns may involve the reuse of small components such as the cells found in both plants or animals, or of larger structures, such as vascular systems, also found in both plants and animals.

5. **Evolution from simple to complex systems**
   Usually, complex systems evolve from simple systems. But it can be evolved much faster if there are stable intermediate forms. During the evolution process, the objects that were complex are considered as the most primitive objects based on which the system is developed.

ORGANIZED AND DISORGANIZED COMPLEXITY s(cannonical form of a complex system)

Simplifying Complex Systems

- Usefulness of abstractions common to similar activities
  e.g. driving different kinds of motor vehicle
- Multiple orthogonal hierarchies
  e.g. structure and control system
- Prominent hierarchies in object-orientation
  “ class structure ”
  “ object structure ”
  e. g. engine types, engine in a specific car

One mechanism to simplify concerns in order to make them more manageable is *to identify and understand abstractions common to similar objects or activities.*

There are two orthogonal hierarchies of system, its class structure and the object structure.

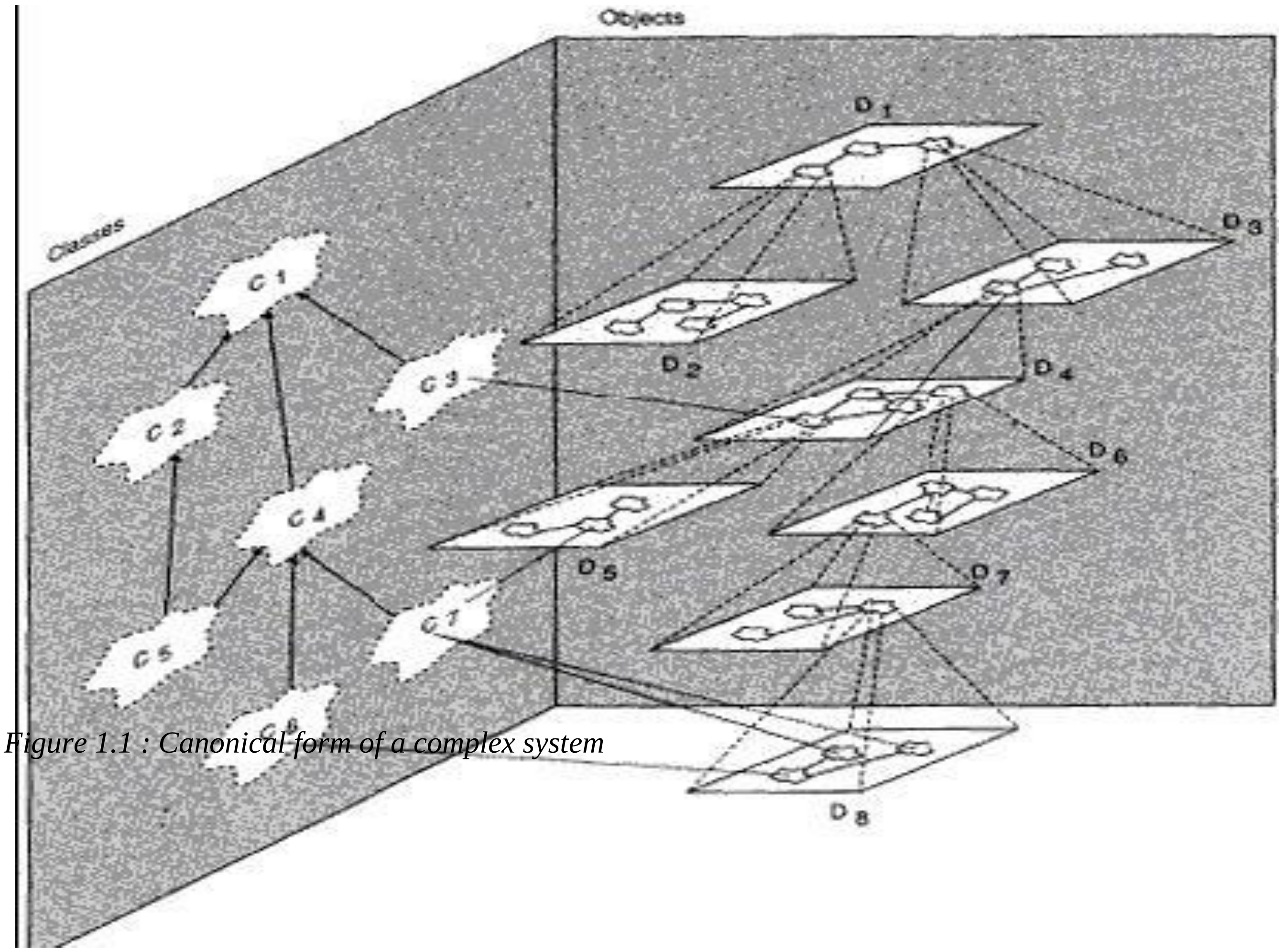

Figure 1.1 : Canonical form of a complex system

The figure 1.1 represents the relationship between two different hierarchies: a hierarchy of objects and a hierarchy of classes. The class structure defines the 'is-a' hierarchy, identifying the commonalities between different classes at different levels of abstractions. Hence class C4 is also a class C1 and therefore has every single property that C1 has. C4, however, may have more specific properties that C1 does not have; hence the distinction between C1 and C4. The object structure defines the 'part-of' representation. This identifies the composition of an object from component objects, like a car is composed from wheels, a steering wheel, a chassis and an engine. The two hierarchies are not entirely orthogonal as objects are instances of certain classes. The relationship between these two hierarchies is shown by identifying the instance-of relationship as well. The objects in component D8 are instances of C6 and C7 As suggested by the diagram, there are many more objects then there are classes. The point in identifying classes is therefore to have a vehicle to describe only once all properties that all instances of the class have.

The Limitations of the human capacity for dealing with complexity:

- Human beings are able to understand and remember fairly complex diagrams, though linear notations expressing the same concepts are not dealt with so easily. This is why many methods rely on diagramming techniques as a basis. The human mind is also rather limited. Miller revealed in 1956 that humans can only remember 7 plus or minus one item at once. Methods should therefore encourage its users to bear these limitations in mind and not deploy overly complex diagrams.

BRINGING ORDER TO CHAOS

Principles that will provide basis for development

- Abstraction
- Hierarchy
- Decomposition

The Role of Abstraction: Abstraction is a powerful technique to handle the complexity. It provides a way to manage the system components and their corresponding interactions without considering the details of building those components. For example, consider a car which is a complex system and has several parts within it. A car contains fe high level components such as brake, handle, accelerator, gears etc. it is possible to describe the use of these components with out giving details about their construction.

When details of the construction of any one of these components is to be found out , the construction of remaining components is not considered, but their relation with their component is considered. This means , only abstract view of a component is taken in to consideration.

Definition: “process of elimination of irrelevant information and displays only essential information.

Object-orientation attempts to deploy abstraction. The common properties of similar objects are defined in an abstract way in terms of a class. Properties that different classes have in common are identified in more abstract classes and then an ‘is-a’ relationship defines the inheritance between these classes.

The role of Hierarchy: Identifying the hierarchies within a complex software system makes understanding of the system very simple. The object structure is important because it illustrates how different objects collaborate with one another through pattern of interaction (called mechanisms). By classifying objects into groups of related abstractions (for example, kinds of plant cells versus animal cells, we come to explicitly distinguish the common and distinct properties of different objects, which helps to master their inherent complexity.

The role of Decomposition: Decomposition is important techniques. At the time of the design of the complex software systems, it is necessary to decompose the software into smaller components This technique is based on the idea of divide and conquer. In dividing a problem into a sub problem the problem becomes less complex and easier to understand and to deal with. Repeatedly dividing a problem will lead to sub problems that are small enough so that they can be conquered. After all the sub problems have been conquered and solutions to them have been found, the solutions need to be composed in order to obtain the solution of the whole problem. There are two types of decomposition techniques 1) process-oriented (Algorithmic) 2) object-oriented decomposition.

Algorithmic (Process Oriented) Decomposition: In Algorithmic decomposition, the building blocks of software are procedures or functions that concentrates on the decomposition process. in this process, the larger algorithms are decomposed into smaller well managed algorithms. the drawback of this approach is that, it is difficult to maintain the software since the requirements are not static and changes depending on the users specifications. here, every individual module with in the system represent main steps in some process.

the diagram below illustrates, the products of structured design. it represents the structured chart describing the relationships between discrete functional elements of the solution. the chart describes some position of the design of a program for updating the content of a master file.

the chart is created by automatic generation through data flow diagram by expert system tool.

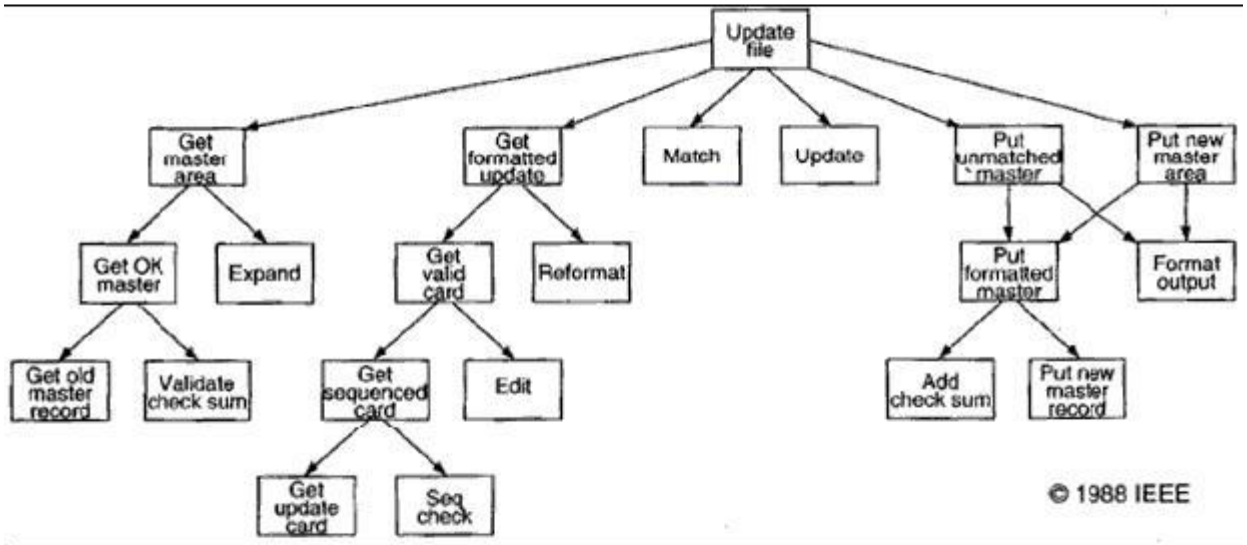

Figure 1.2: Algorithmic decomposition

Object oriented decomposition:

In this approach. decomposition is based on objects and not algorithms. Objects are identified as Master file and check sum which derive directly from the vocabulary of the problem as shown in figure. Here, get formatted update is not a separate algorithm but it can be described as an operation associated with object file of updates. When this operation is called, another object called update to card gets created.

There fore . this solution entails each object to express its own unique behavior and also every individual objects models another object in the real world.

According to this an object can be simply defined as an entity which specifies well defined behavior. The users are responsible in making the objects to do things, through sending the messages. Since decomposition is based on objects , hence it is reffered to as object oriented decomposition.

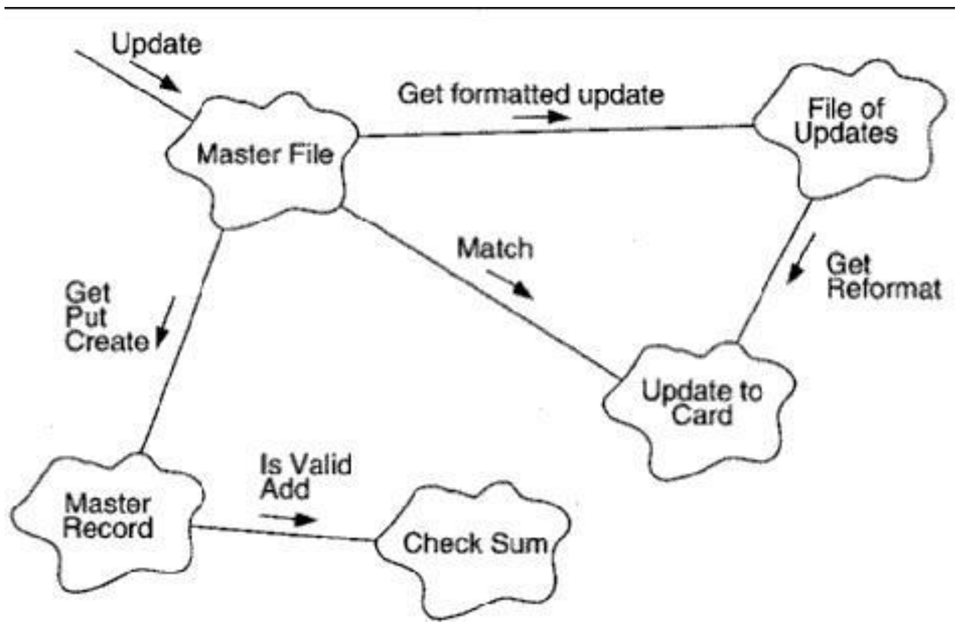

Figure 1.3: Object Oriented decomposition

Algorithmic versus object oriented decomposition: The algorithmic view highlights the ordering of events and the object oriented view emphasizes the agents that either cause action or are the subjects upon which these operations act. We must start decomposing a system either by algorithms or by objects then use the resulting structure as the framework for expressing the other perspective generally object oriented view is applied because this approach is better at helping us organize the inherent complexity of software systems. object oriented algorithm has a number of advantages over algorithmic decomposition. Object oriented decomposition yields smaller systems through the reuse of common mechanisms, thus providing an important economy of expression and are also more resident to change and thus better able to involve over time and it also reduces risks of building complex software systems. Object oriented decomposition also directly addresses the inherent complexity of software by helping us make intelligent decisions regarding the separation of concerns in a large state space.

Process-oriented decompositions divide a complex process, function or task into simpler sub processes until they are simple enough to be dealt with. The solutions of these sub functions then need to be executed in certain sequential or parallel orders in order to obtain a solution to the complex process. Object-oriented decomposition aims at identifying individual autonomous objects that encapsulate both a state and a certain behavior. Then communication among these objects leads to the desired solutions.

Although both solutions help dealing with complexity we have reasons to believe that an object-oriented decomposition is favorable because, the object-oriented approach provides for a semantically richer framework that leads to decompositions that are more closely related to entities from the real world. Moreover, the identification of abstractions supports (more abstract) solutions to be reused and the object-oriented approach supports the evolution of systems better as those concepts that are more likely to change can be hidden within the objects.

DESIGNING COMPLEX SYSTEMS

The science and arts are two elements that play critical role in every branch of engineering. Ideally, the concept of design that corresponds to new structure which involves exceptional imagination, experience, knowledge. This is expected by any artist engineer working in field. After the design is expressed by the engineer as an artist, it is then analyzed by engineer as scientist. i.e the programming challenge can be specified.

Engineering as a Science and an Art: the role of an engineer as an artist is significantly important because their task is to design completely new systems.

Every engineering discipline involves elements of both science and art. The programming challenge is a large scale exercise in applied abstraction and thus requires the abilities of the formal mathematician blended with the attribute of the competent engineer.

The meaning of Design: The purpose of design is to develop a clean and simple internal structure which is referred to a as architecture.

1. It should Satisfy a given functional specifications
2. Conforms to limitations of the target medium (it should be guided by te constraints of target medium)

3. Meets implicit or explicit requirements on performance and resource usage
4. Satisfies implicit or explicit design criteria on the form of the artifact

5. Satisfies restrictions on the design process itself, such as its length or cost, or the available for doing the design.

According to Stroustrup, the purpose of design is to create a clean and relatively simple internal structure, sometimes also called as architecture. A design is the end product of the design process.

The Importance of Model Building:

The building of models has a broad acceptance among all engineering disciplines, largely because model building appeals to the principles of decomposition, abstraction, and hierarchy. Development of new models is based upon the old models. each model can be evaluated under expected and unexpected conditions. And later, they can be altered if they fail to perform as expected.

The Elements of Software design Methods: Design of complex software system involves an incremental and iterative process. Each method includes the following:

1. **Notation:** The language for expressing each model.

2. **Process:** The activities leading to the orderly construction of the system's mode.
3. **Tools:** It represents the artifacts **that eliminates the** tediousness of the model development . it also applies rules in order to identify errors and inconsistencies in the system.

The models of Object Oriented Development: model building plays an important role in the development of complex systems. The model under the object oriented analysis and design specifies the significance of class ad object hierarchies of the systems.

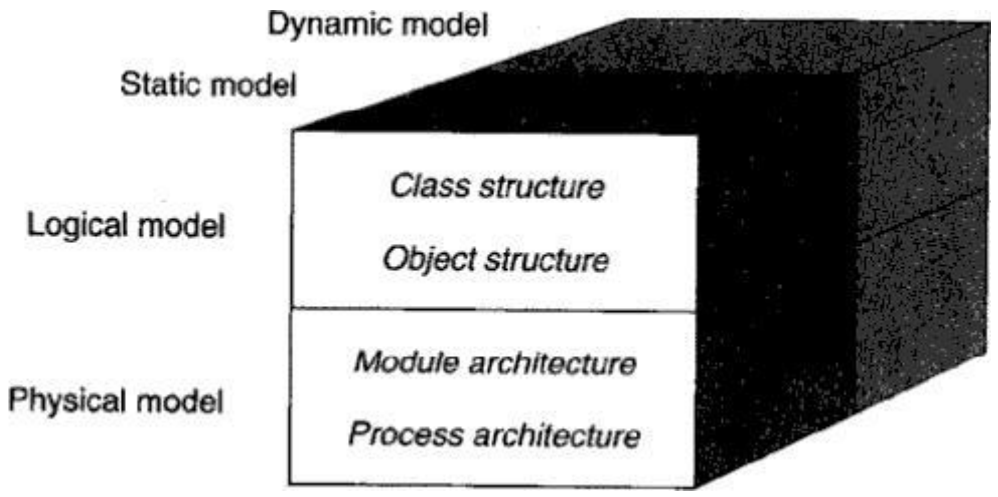

Figure 1.4: Models of object oriented development

Booch presents a model of object-oriented development that identifies several relevant perspectives. The classes and objects that form the system are identified in a logical model. For this logical model, again two different perspectives have to be considered. A static perspective identifies the structure of classes and objects, their properties and the relationships classes and objects participate in. A dynamic model identifies the dynamic behavior of classes and objects, the different valid states they can be in and the transitions between these states.

Besides the logical model, also a physical model needs to be identified. This is usually done later in the system's lifecycle. The module architecture identifies how classes are kept in separately compliable modules and the process architecture identifies how objects are distributed at run-time over different operating system processes and identifies the relationships between those. Again for this physical model a static perspective is defined that considers the structure of module and process architecture and a dynamic perspective identifies process and object activation strategies and inter-process communication. Object-orientation has not, however, emerged fully formed. In fact it has developed over a long period, and continues to change.

The Object Model

The elements of the object oriented technology collectively known as the object model. The object model encompasses the principles of abstraction, encapsulation, modularity, hierarchy, typing, concurrency and persistency. The object model brought together these elements in a synergistic way.

THE EVOLUTION OF THE OBJECT MODEL

- The shift in focus from programming-in-the-small to programming-in-the-large.
- The evolution of high-order programming languages.
- New industrial strength software systems are larger and more complex than their predecessors.
- Development of more expressive programming languages advances the decomposition, abstraction and hierarchy.
- Wegner has classified some of more popular programming languages in generations according to the language features.

The generation of programming languages

1. First generation languages (1954 – 1958)
   - Used for specific & engineering application.
   - Generally consists of mathematical expressions.
   - For example: FORTRAN I, ALGOL 58, Flowmatic, IPLV etc.
2. Second generation languages (1959 – 1961)
   - Emphasized on algorithmic abstraction.
   - FORTRAN II - having features of subroutines, separate compilation

- ALGOL 60 - having features of block structure, data type

- COBOL - having features of data, descriptions, file handing
- LISP - List processing, pointers, garbage collection

3. Third generation languages (1962 – 1970)
    - Supports data abstraction.

    - PL/1 – FORTRAN + ALGOL + COBOL
    - ALGOL 68 – Rigorous successor to ALGOL 60

    - Pascal – Simple successor to ALGOL 60
    - Simula - Classes, data abstraction

4. The generation gap (1970 – 1980)
    - C – Efficient, small executables

    - FORTRAN 77 – ANSI standardization

5. Object Oriented 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

6. 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 – VB for Microsoft .NET framework

Topology of first and early second generation programming languages

- Topology means basic physical building blocks of the language & how those parts can be connected.
- Arrows indicate dependency of subprograms on various data.
- Error in one part of program effect across the rest of system.

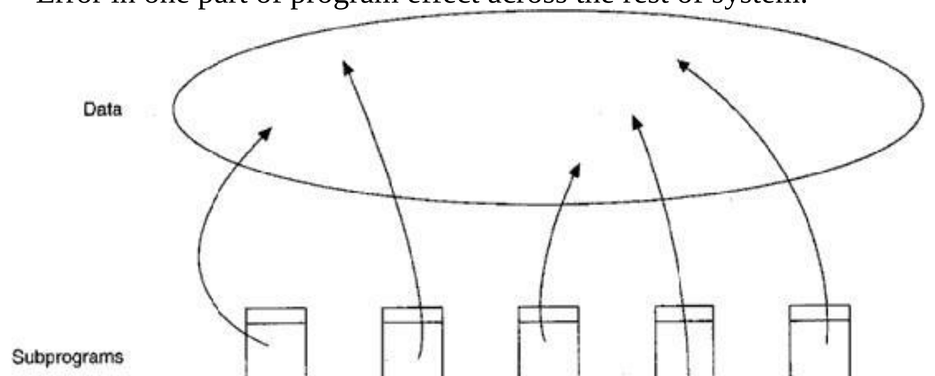

Fig 2.1: The Topology of First- and Early Second-Generation Programming Languages

Topology of late second and early third generation programming languages
Software abstraction becomes procedural abstraction; subprograms as an obstruction mechanism and three important consequences:
- Languages invented that supported parameter passing mechanism
- Foundations of structured programming were laid.
- Structured design method emerged using subprograms as basic physical blocks.

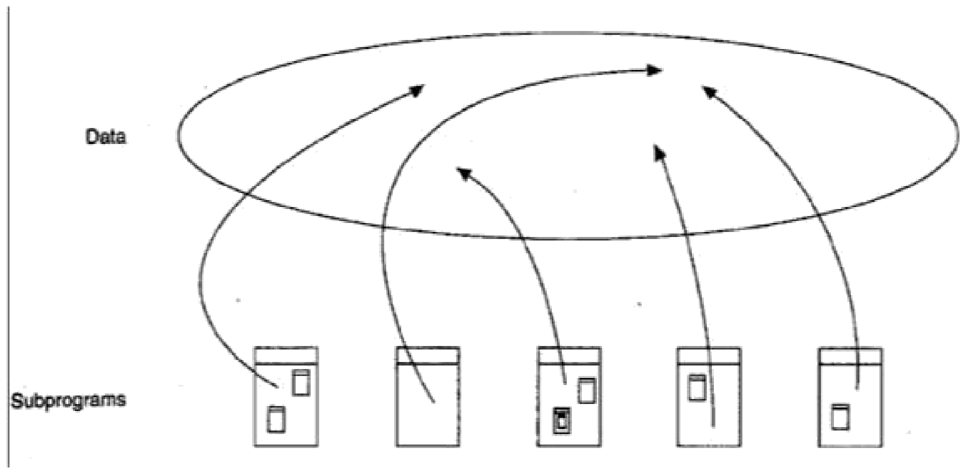

Fig 2.2: The Topology of Late Second- and Early Third-Generation Programming Languages

The topology of late third generation programming languages

- Larger project means larger team, so need to develop different parts of same program independently, i.e. complied module.
- Support modular structure.

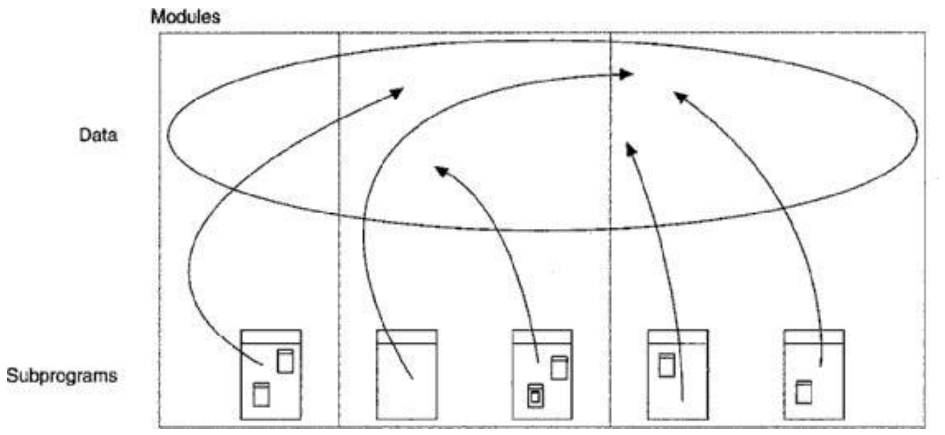

Fig 2.3: The Topology of Late Third-Generation Programming Languages

Topology of object and object oriented programming language Two methods for complexity of problems

(i) Data driven design method emerged for data abstraction.

(ii) Theories regarding the concept of a type appeared

- Many languages such as Smalltalk, C++, Ada, Java were developed.
- Physical building block in these languages is module which represents logical collection of classes and objects instead of subprograms.
- Suppose procedures and functions are verbs and pieces of data are nouns, then
- Procedure oriented program is organized around verbs and object oriented program is organized around nouns.
  Data and operations are united in such a way that the fundamental logical building blocks of our systems are no longer algorithms, but are classes and objects.
    - In large application system, classes, objects and modules essential yet insufficient means of abstraction.

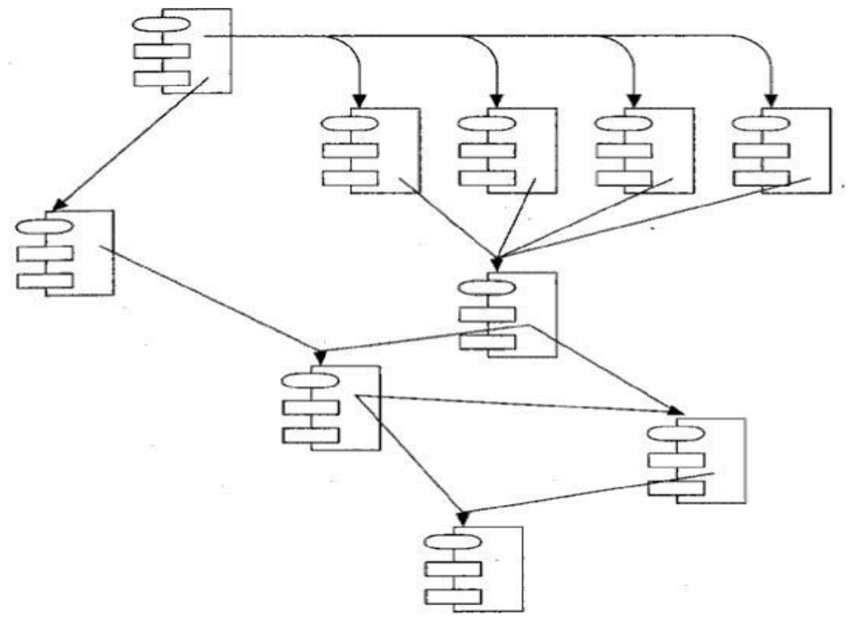

Fig 2.4: The Topology of Small- to Moderate-Sized Applications Using Object-Based and Object-Oriented Programming Languages

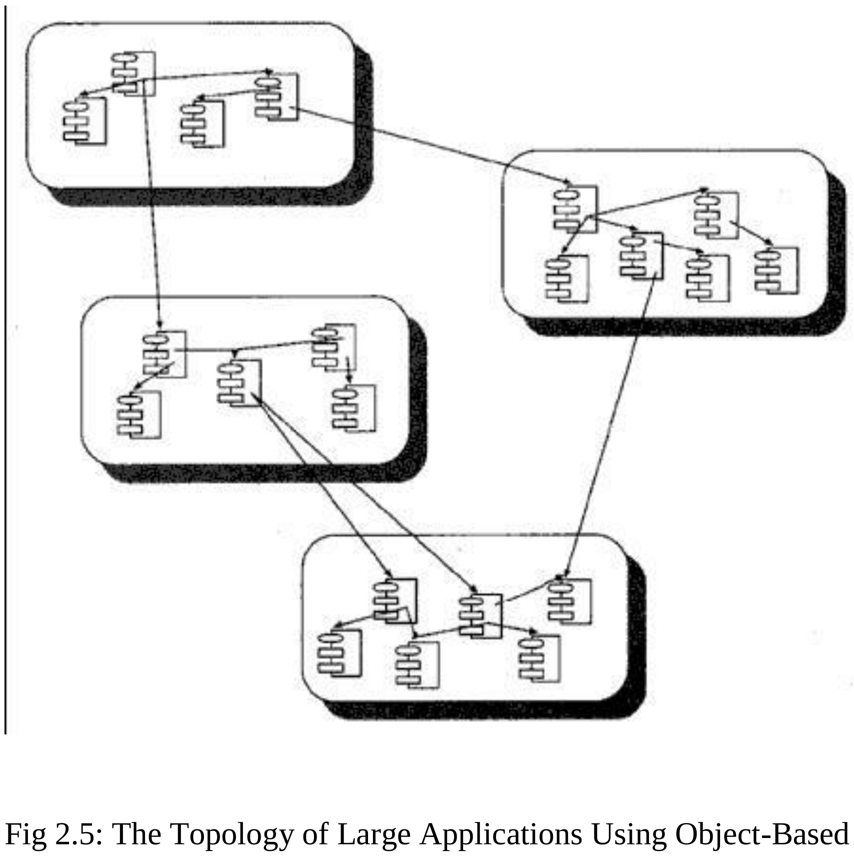

Fig 2.5: The Topology of Large Applications Using Object-Based and Object-Oriented Programming Languages

FOUNDATIONS OF THE OBJECT MODEL

In structured design method, build complex system using algorithm as their fundamental building block. An object oriented programming language, class and object as basic building block.
Following events have contributed to the evolution of object-oriented concepts:

- Advances in computer architecture, including capability systems and hardware support for operating systems concepts
- Advances in programming languages, as demonstrated in Simula, Smalltalk, CLU, and Ada.
- Advances in programming methodology, including modularization and information

hiding. We would add to this list three more contributions to the foundation of the object model:

- Advances in database models
- Research in artificial intelligence
- Advances in philosophy and cognitive science

OOA (Object Oriented analysis)

During software requirement phase, requirement analysis and object analysis, it is a method of analysis that examines requirements from the perspective of classes and objects as related to problem domain. Object oriented analysis emphasizes the building of real-world model using the object oriented view of the world.

OOD (Object oriented design)

During user requirement phase, OOD involves understanding of the application domain and build an object model. Identify objects; it is methods of design showing process of object oriented decomposition. 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.

OOP (Object oriented programming)

During system implementation phase, t 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 in inheritance relationships. Object oriented programming satisfies the following requirements:

- It supports objects that are data abstractions with an interface of named operations and a hidden local state.
- Objects have associated type (class).
- Classes may inherit attributes from supertype to subtype.

ELEMENTS OF OBJECT MODEL

Kinds of Programming Paradigms: According to Jenkins and Glasgow, most programmers work in one language and use only one programming style. They have not been exposed to alternate ways of thinking about a problem. Programming style is 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.
There are five main kinds of programming styles:

1. Procedure oriented – Algorithms for design of computation
2. Object oriented – classes and objects
3. Logic oriented – Goals, often expressed in a predicate calculus
4. Rules oriented – If then rules for design of knowledge base
5. Constraint orient – Invariant relationships.

Each requires a different mindset, a different way of thinking about the problem. Object model is the conceptual frame work for all things of object oriented.

There are four **major elements** of object model. They are:

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

There are three **minor elements** which are useful but not essential part of object model. Minor elements of object model are:

1. Typing
2. Concurrency
3. Persistence

Abstraction

Abstraction is defined as a simplified description or specification of a system that emphasizes some of the system details or properties while suppressing others. A good abstraction is one that emphasizes details that are significant to the reader or user and suppresses details that are, not so significant, immaterial.
An abstraction denotes the essential characteristics of an object that distinguishes it from all other kinds of objects and thus provides crisply defined conceptual boundaries on the perspective of the viewer. An abstraction focuses on the outside view of an object, Abstraction focuses up on the essential characteristics of some object, relative to the perspective of the viewer. From the most to the least useful, these kinds of abstraction include 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 program the same kind of function.
- Virtual machine abstractions: An object that groups together operations that are used by some superior level of control, or operations that all use some junior set of operations.
- Coincidental abstraction: An object that packages a set of operations that have no relation to each other.

Abstraction: Temperature Sensor
Important Characteristics:
Temperature location

Figure 2.6: Abstraction of a Temperature Sensor.

Modularity

The act of partitioning a program into individual components is called modularity. It is reusable component which reduces complexity to some degree. Although partitioning a program is helpful for this reason, a more powerful justification for partitioning a program is that it creates a number of well-defined, documented boundaries within the program. These boundaries, or interfaces, are invaluable in the comprehension of the program. In some languages, such as Smalltalk, there is no concept of a module, so the class forms the only physical unit of decomposition. Java has packages that contain classes. In many other languages, including Object Pascal, C++, and Ada, the module is a separate language construct and therefore warrants a separate set of design decisions. In these languages, classes and objects form the logical structure of a system; we place these abstractions in modules to produce the system's physical architecture. Modularization consists of dividing a program into modules which can be compiled separately, but which have connections with other modules. Modularity is the property of a system that has been decomposed into a set of cohesive and loosely coupled modules.

- modules can be compiled separately. modules in C++ are nothing more than separately compiled files, generally called header files.
- Interface of module are files with .h extensions & implementations are placed in files with

  .c or .cpp suffix.

- Modules are units in pascal and package body specification in ada.

- modules Serve as physical containers in which classes and objects are declared like gates in IC of computer.

- Group logically related classes and objects in the same module.

- E.g. consider an application that runs on a distributed set of processors and uses a message passing mechanism to coordinate their activities.

- A poor design is to define each message class in its own module; so difficult for users to find the classes they need. Sometimes modularization is worse than no modulation at all.

- Developer must balance: desire to encapsulate abstractions and need to make certain abstractions visible to other modules.

- Principles of abstraction, encapsulation and modularity are synergistic (having common effect)

Example of modularity

Let's look at modularity in the Hydroponics Gardening System. Suppose we decide to use a commercially available workstation where the user can control the system's operation. At this workstation, an operator could create new growing plans, modify old ones, and follow the progress of currently active ones. Since one of our key abstractions here is that of a growing plan, we might therefore create a module whose purpose is to collect all of the classes associated with individual growing plans (e.g., FruitGrowingPlan, GrainGrowingPlan). The implementations of these GrowingPlan classes would appear in the implementation of this module. We might also define a module whose purpose is to collect all of the code associated with all user interface functions.

Hierarchy

Hierarchy is a ranking or ordering of abstractions Encapsulation hides company inside new of abstraction and modularity logically related abstraction & thus a set of abstractions form hierarchy. Hierarchies in complex system are its class structure (the "is a" hierarchy) and its object structure (the "part of" hierarchy).

Examples of Hierarchy: Single Inheritance

Inheritance defines a relationship among classes. Where one classes shares structure or behaviors defined in one (single in heritance) or more class (multiple inheritance) & thus represents a hierarchy of abstractions in which a subclass inherits from one or more super classes. Consider the different kinds of growing plans we might use in the Hydroponics Gardening System. An earlier section described our abstraction of a very generalized growing plan. Different kinds of crops, however, demand specialized growing plans. For example, the growing plan for all fruits is generally the same but is quite different from the plan for all vegetables, or for all floral crops. Because of this clustering of abstractions, it is reasonable to define a standard fruitgrowing plan that encapsulates the behavior common to all fruits, such as the knowledge of when to pollinate or when to harvest the fruit. We can assert that FruitGrowingPlan "is a" kind of GrowingPlan.

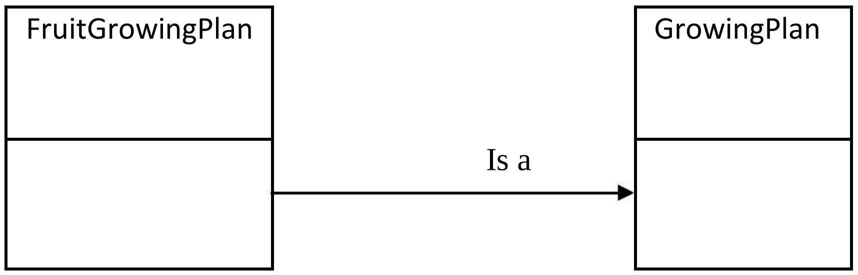

Fig 2.8: Class having one superclass (Single Inheritance)

In this case, FruitGrowingPlan is more specialized, and GrowingPlan is more general. The same

could be said for GrainGrowingPlan or VegetableGrowingPlan, that is, GrainGrowingPlan "is a" kind of Growin

gPlan, and VegetableGrowingPlan "is a" kind of GrowingPlan. Here,

GrowingPlan is the more general superclass, and the others are specialized subclasses.

As we evolve our inheritance hierarchy, the structure and behavior that are common for different classes will tend to migrate to common superclasses. This is why we often speak of inheritance as being a generalization/specialization hierarchy. Superclasses represent generalized abstractions, and subclasses represent specializations in which fields and methods from the superclass are added, modified, or even hidden

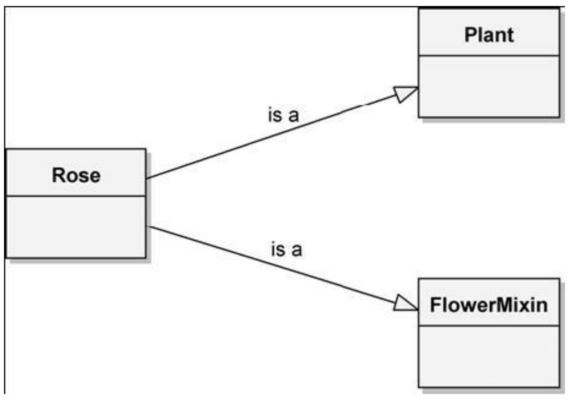

Figure 2.9: The Rose Class, Which Inherits from Multiple Super classes (Multiple Inheritance)

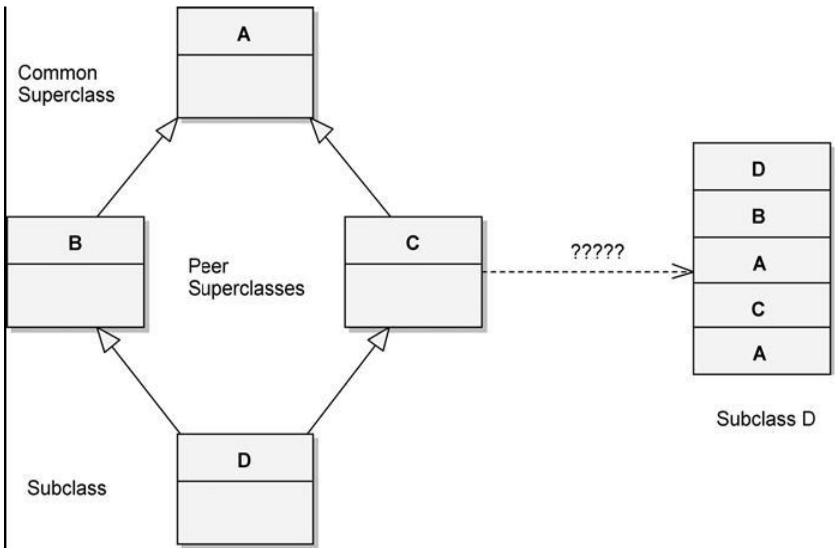

Figure 2.10: The Repeated Inheritance

Repeated inheritance occurs when two or more peer super classes share a common super class.

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.

Aggregation raises the issue of ownership. Our abstraction of a garden permits different plants to be raised in a garden over time, but replacing a plant does not change the identity of the garden as a whole, nor does removing a garden necessarily destroy all of its plants (they are likely just transplanted). In other words, the lifetime of a garden and its plants are independent. In contrast, we have decided that a Growing Plan object is intrinsically associated with a Garden object and does not exist independently. Therefore, when we create an instance of Garden, we also create an instance of Growing Plan; when we destroy the Garden object, we in turn destroy the Growing Plan instance.

Typing

A type is a precise characterization of structural or behavioral properties which a collection of entities share. Type and class are used interchangeably class implements a type. Typing is the enforcement of the class of an object. Such that object of different types may not be interchanged. Typing implements abstractions to enforce design decisions. E.g. multiplying temp by a unit of force does not make serve but multiplying mass by force does. So this is strong typing. Example of strong and weak typing: In strong type, type conformance is strictly enforced. Operations can not be called upon an object unless the exact signature of that operation is defined in the object's class or super classes.

A given programming language may be strongly typed, weakly typed, or even untyped, yet still be called object-oriented. A strongly typed language is one in which all expressions defined in super class are guaranteed to be type consistent. When we divide distance by time, we expect some value denoting speed, not weight. Similarly, dividing a unit of force by temperature doesn’t make sense, but dividing force by mass does. These are both examples of strong typing, wherein the rules of our domain prescribe and enforce certain legal combinations of abstractions.

Benefits of Strongly typed languages:

- Without type checking, program can crash at run time
- Type declaration help to document program
- Most compilers can generate more efficient object code if types are declared.

Examples of Typing: Static and Dynamic Typing

Static typing (static binding/early binding) refers to the time when names are bound to types i.e. types of all variables are fixed at the time of compilation. Dynamic binding (late binding) means that types of all variables and expressions are not known until run time. Dynamic building (object pascal, C++) small talk (untyped).

Polymorphism is a condition that exists when the features of dynamic typing and inheritance interact. Polymorphism represents a concept in type theory in which a single name (such as a variable declaration) may denote objects of many different classes that are related by some common super class. The opposite of polymorphism is monomorphism, which is found in all languages that are both strongly and statically typed.

Concurrency

OO-programming focuses upon data abstraction, encapsulation and inheritance concurrency focuses upon process abstraction and synchronization. Each object may represent a separate thread of actual (a process abstraction). Such objects are called active. In a system based on an object oriented design, we can conceptualize the word as consisting of a set of cooperative objects, some of which are active (serve as centers of independent activity). Thus concurrency is the property that distinguishes an active object from one that is not active. For example: If two active objects try to send messages to a third object, we must be certain to use some means of mutual exclusion, so that the state of object being acted upon is not computed when both active objects try to update their state simultaneously. In the preserve of concurrency, it is not enough simply to define the methods are preserved in the presence of multiple thread of control.

Examples of Concurrency

Let's consider a sensor named Active Temperature Sensor, whose behavior requires periodically sensing the current temperature and then notifying the client whenever the temperature changes a certain number of degrees from a given set point. We do not explain how the class implements this behavior. That fact is a secret of the implementation, but it is clear that some form of concurrency is required.

There are three approaches to concurrency in object oriented design\

- Concurrency is an intrinsic feature of languages. Concurrency is termed as task in ada, and class process in small talk. class process is used as super classes of all active objects. we may create an active object that runs some process concurrently with all other active objects.

- We may use a class library that implements some form of light weight process AT & T task library for C++ provides the classes' sched, Timer, Task and others. Concurrency appears through the use of these standard classes.

- Use of interrupts to give the illusion of concurrency use of hardware timer in active Temperature sensor periodically interrupts the application during which time all the sensor read the current temperature, then invoke their callback functions as necessary.

Persistence

Persistence is the property of an object through which its existence transcends time and or space objects continues to exist after its creator ceases to exist and/or the object's location moves from the address space in which it was created. An object in software takes up some amount of space and exists for a particular amount of time. Object persistence encompasses the followings.

- Transient results in expression evaluation
- Local variables in procedure activations
- Global variables where exists is different from their scope
- Data that exists between executions of a program
- Data that outlines the Program.

Traditional Programming Languages usually address only the first three kind of object persistence. Persistence of last three kinds is typically the domain of database technology. Introducing the concept of persistence to the object model gives rise to object oriented databases. In practice, such databases build upon some database models (Hierarchical, network relational). Database queries and operations are completed through the programmer abstraction of an object oriented interface. Persistence deals with more than just the lifetime of data. In object oriented databases, not only does the state of an object persist, but its class must also transcend only individual program, so that every program interprets this saved state in the same way.

In most systems, an object once created, consumes the same physical memory until it classes to exist. However, for systems that execute upon a distributed set of processors, we must sometimes be concerned with persistence across space. In such systems, it is useful to think of objects that can move from space to space.

APPLYING THE OBJECT MODEL

Benefits of the Object Model: Object model introduces several new elements which are advantageous over traditional method of structured programming. The significant benefits are:

- Use of object model helps us to exploit the expressive power of object based and object oriented programming languages. Without the application of elements of object model, more powerful feature of languages such as C++, object pascal, ada are either ignored or greatly misused.
- Use of object model encourages the reuse of software and entire designs, which results in the creation of reusable application framework.
- Use of object model produces systems that are built upon stable intermediate forms, which are more resilent to change.
- Object model appears to the working of human cognition, many people who have no idea how a computer works find the idea of object oriented systems quite natural.

OOA & Design may be in only method which can be employed to attack the complexity inherent in large systems. Some of the applications of the object model are as follows:

- Air traffic control
- Animation
- Business or insurance software
- Business Data Processing
- CAD
- Databases
- Expert Systems
- Office Automation
- Robotics
- Telecommunication
- Telemetry System etc