NARAYANA ENGINEERING COLLEGE::GUDUR

Dhurjati Nagar-524101

DEPARTMENT OF MCA

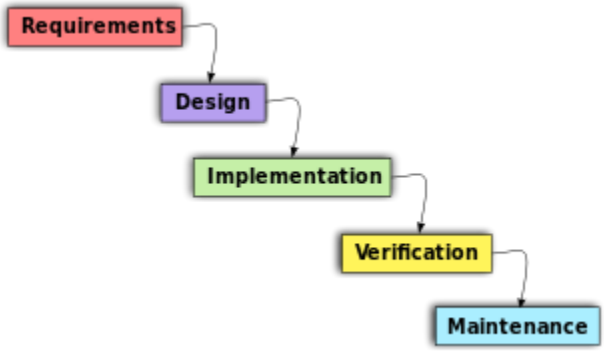
II MCA-IV Semester: **SUB: OBJECT ORIENTED ANALYSIS & DESIGN**

UNIT-1

Introduction: 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.

Introduction: It's a structured method for analyzing, designing a system by applying the object-orientated concepts, and develop a set of graphical system models during the development life cycle of the software.

OOAD In The SDLC: The software life cycle is typically divided up into stages going from abstract descriptions of the problem to designs then to code and testing and finally to deployment.



The earliest stages of this process are analysis (requirements) and design.The distinction between analysis and design is often described as "what Vs how".

In analysis developers work with users and domain experts to define what the system is supposed to do. Implementation details are supposed to be mostly or totally ignored at this phase.

The goal of the analysis phase is to create a model of the system regardless of constraints such as appropriate technology. This is typically done via use cases and *abstract* definition of the most important objects using conceptual model.

The design phase refines the analysis model and applies the needed technology and other implementation constrains.

It focuses on describing the objects, their attributes, behavior, and interactions. The design model should have all the details required so that programmers can implement the design in code.

Object-Oriented Analysis:

In the object-oriented analysis, we …

1. **Elicit** *requirements:* Define what does the software need to do, and what's the problem the software trying to solve.
2. **Specify** *requirements*: Describe the requirements, usually, using use cases (and scenarios) or user stories.
3. **Conceptual** *model*: Identify the important objects, refine them, and define their relationships and behavior and draw them in a simple diagram.

Object-Oriented Design:

The analysis phase identifies the objects, their relationship, and behavior using the conceptual model (an **abstract** definition for the objects).

While in design phase, we describe these objects (by creating class diagram from conceptual diagram—usually mapping conceptual model to class diagram), their attributes, behavior, and interactions.

In addition to applying the software design principles and patterns which will be covered in later tutorials.

The input for object-oriented design is provided by the output of object-oriented analysis. But, analysis and design may occur in parallel, and the results of one activity can be used by the other.

In the object-oriented design, we …

1. **Describe the classes** and their relationships using class diagram.
2. **Describe the interaction** between the objects using sequence diagram.
3. **Apply** software design principles and design patterns.

   A class diagram gives a visual representation of the classes you need. And here is where you get to be really specific about object-oriented principles like inheritance and polymorphism.

Describing the interactions between those objects lets you better understand the responsibilities of the different objects, the behaviors they need to have.

The Structure of Complex systems:

Definition: A complex system is a system composed of many components which may interact with each other. Examples of complex systems are Earth's global climate, organisms, the human brain, infrastructure such as power grid, transportation or communication systems, social and economic organizations.

Systems: Systems are constructed by interconnecting components (Boundaries, Environments, Characters, Emergent Properties), which may well be systems in their own right. The larger the number of these components and relationships between them, higher will be the complexity of the overall system.

Software Systems: Software systems are not any different from other systems with respect to these characteristics. Thus, they are also embedded within some operational environment, and perform operations.

Thus, they are also embedded within some operational environment, and perform operations which are clearly defined and distinguished from the operations of other systems in this environment.

They also have properties which emerge from the interactions of their components and/or the interactions of themselves with other systems in their environment.

A system that embodies one or more software subsystems which contribute to or control a significant part of its overall behavior is what we call a software intensive system.

As examples of complex software-intensive systems, we may consider stock and production control systems, aviation systems, rail systems, banking systems, health care systems and 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 which carry;

Impossible for humans to comprehend fully
- Difficult to document and test
- Potentially inconsistent or incomplete
- Subject to change
- No fundamental laws to explain phenomena and approaches .

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

1. **The structure of a Personal Computer:**

   A personal computer is a device of moderate complexity. Major elements are CPU, monitor, keyboard and some secondary storage devices.

CPU encompasses primary memory, an ALU, and a bus to which peripheral devices are attached.

An ALU may be divided into registers which are constructed from NAND gates, inverters and so on. All are the hierarchical nature of a complex system.

- Computer System
    - CPU, Input, Output
        - ALU, CU, Registers
            - Logic Gates
    - we can decompose, in hierarchy, and use different level of abstraction

2. **The structure of Plants:**

Plants are complex multicellular organism which are composed of cells which is turn encompasses elements such as chloroplasts, nucleus, and so on.

For example, at the highest level of abstraction, roots are responsible for absorbing water and minerals from the soil.

Roots interact with stems, which transport these raw materials up to the leaves. The leaves in turn use water and minerals provided by stems to produce food through photosynthesis.

3. **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.

4. **The structure of Matter:**

Nuclear physicists are concerned with a structural hierarchy of matter. Atoms are made up of electrons, protons and neutrons.

Elements and elementary particles but protons, neutrons and other particles are formed from more basic components called quarks, which eventually formed from pro-quarks.

- Galaxy to Atoms are governed by same forces such as gravity and electromagnetic forces

5. **The structure of Social institutions:**

In social institutions, group of people join together to accomplish tasks that cannot be done by made of divisions which in turn contain branches which in turn encompass local offices and so on

- Organizations
  - Branches
    - Departments

    Offices

The Inherent Complexity of Software:

Definition: 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.

The Properties of Complex and Simple Software Systems: Software may involve elements of great complexity which is of different kind.

Some software systems are simple.

- These are the largely forgettable applications that are specified, constructed, maintained, and used by the same person, usually the amateur programmer or the professional developer working in isolation.
- Such systems tend to have a very limited purpose and a very short life span.
- We can afford to throw them away and replace them with entirely new software rather than attempt to reuse them, repair them, or extend their functionality, Such applications are generally more tedious than difficult to develop; consequently, learning how to design them does not interest us.

Some software systems are complex.

- The applications that exhibit a very rich set of behaviors, as, for example, in reactive systems that drive or are driven by events in the physical world, and for which time and space are scarce resources; applications that maintain the integrity of hundreds of thousands of records of information while allowing concurrent updates and queries.
- Systems for the command and control of real-world entities, such as the routing of air or railway traffic.
- Software systems such as world of industrial strength software tend to have a long life span, and over time, many users come to depend upon their proper functioning.
- The frameworks that simplify the creation of domain-specific applications, and programs that mimic some aspect of human intelligence.
- Although such applications are generally products of research and development they are no less complex, for they are the means and artifacts of incremental and exploratory development.

Attributes of Complex System:

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

1. **Hierarchical and interacting subsystems:(Hierarchical structure)**

   Frequently, complexity takes the form of a hierarchy, whereby a complex system is composed of interrelated subsystems that have in turn their own subsystems and so on, until some lowest level of elementary components is reached.

2. **Arbitrary determination of primitive components:(Relative Primitives)**

   The choice of what components in a system are primitive is relatively arbitrary and is largely up to the discretion of 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:(Separation of concerns)**

   Intra-component linkages are generally stronger than inter-component linkages. This fact has the involving the high frequency dynamics of the components-involving the internal structure of the components – from the low frequency dynamic involving interaction among components.

4. **Combine and arrange common rearranging subsystems:(common patterns)**

   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:(Stable intermediate forms)**

   A complex system that works is invariably bound to have evolved from a simple system that worked …….. A complex system designed from scratch never works and can't be patched up to make it work. You have to start over, beginning with a working simple system.

Booch has identified five properties that architectures of complex software systems have in common.

Firstly, every complex system is decomposed into a hierarchy of subsystems. This decomposition is essential in order to keep the complexity of the overall system manageable.

These subsystems, however, are not isolated from each other, but interact with each other.

Organized and Disorganized Complexity:

The discovery of common abstractions and mechanisms greatly facilitates our understanding of complex systems.

For example, with just a few minutes of orientation, an experienced pilot can step into a multiengine jet aircraft he or she has never flown before and safely fly the vehicle.

Having recognized the properties common to all such aircraft, such as the functioning of the rudder, ailerons, and throttle, the pilot primarily needs to learn what properties are unique to that particular aircraft.

If the pilot already knows how to fly a given aircraft, it is far easier to learn how to fly a similar one.

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. We can use a car as an example (which are considerable complex systems).

Understanding common abstractions in this particular example would, for instance, involve the insight that clutch, accelerator and brakes facilitate the use of a wide range of devices, namely transport vehicles depending on transmission of power from engine to wheels).

Another principle to understand complex systems is the separation of concerns leading to multiple hierarchies that are orthogonal to each other.

In the car example, this could be, for instance, the distinction between physical structure of the car (chassis, body, engine), functions the car performs (forward, back, turn) and control systems the car has (manual, mechanical, and electrical).

In object-orientation, the class structure and the object structure relationship is the simplest form of related hierarchy. It forms a canonical representation for object oriented analysis.

The canonical form of a complex system – the discovery of common abstractions and mechanisms greatly facilitates are standing of complex system.

For example, if a pilot already knows how to fly a given aircraft, it is easier to know how to fly a similar one.

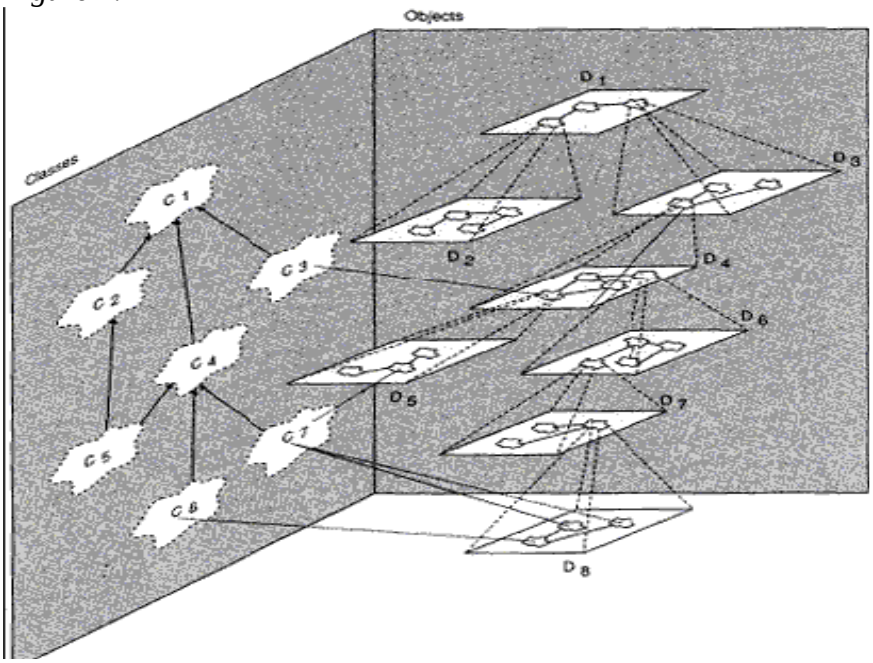
May different hierarchies are present within the complex system. For example an aircraft may be studied by decomposing it into its propulsion system.

Flight control system and so on the decomposition represent a structural or "part of" hierarchy. The complex system also includes an "Is A" hierarchy.

These hierodules for class structure and object structure combining the concept of the class and object structure together with the five attributes of complex system,

virtually all complex system take on the same (canonical) form as shown in figure. There are two orthogonal hierarchies of system, its class structure and the object structure.

Figure 1.1



Canonical form of a complex system

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.

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.

Bringing Order to Chaos:

Definition: Certainly, there will always be geniuses among us, people of extraordinary skill who can do the work of a handful of mere mortal developers, the software engineering equivalents of Frank Lloyd Wright or Leonardo da Vinci. These are the people whom we seek to deploy as our system architects: the ones who devise innovative idioms, mechanisms, and frameworks that others can use as the architectural foundations of other applications or systems.

Principles that will provide basis for development

- Abstraction
- Hierarchy
- Decomposition

Abstraction : Abstraction is an exceptionally powerful technique for dealing with complexity. Unable to master the entirely of a complex object, we choose to ignore its inessential details, dealing instead with the generalized, idealized model of the object.

For example, when studying about how photosynthesis works in a plant, we can focus upon the chemical reactions in certain cells in a leaf and ignore all other parts such as roots and stems. Objects are abstractions of entities in the real world.

In general abstraction assists people's understanding by grouping, generalizing and chunking information.

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.

Different hierarchies support the recognition of higher and lower orders. A class high in the 'is-a' hierarchy is a rather abstract concept and a class that is a leaf represents a fairly concrete concept.

The 'is-a' hierarchy also identifies concepts, such as attributes or operations, that are common to a number of classes and instances.

Decomposition: Decomposition is important techniques for copying with complexity based on the idea of divide and conquer.

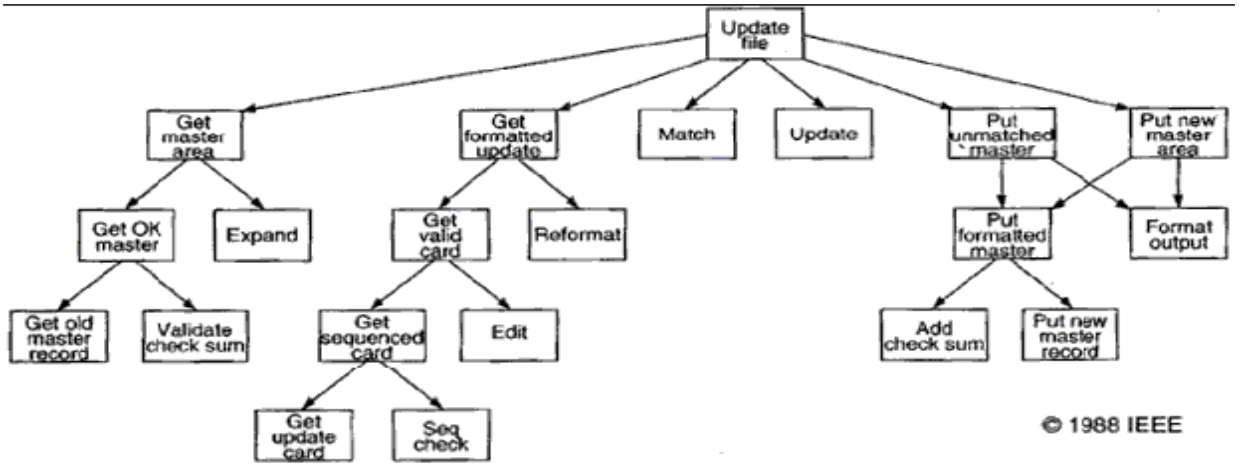
In dividing a problem into a sub problem the problem becomes less complex and easier to overlook and to deal with.

Repeatedly dividing a problem will eventually 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

The history of computing has seen two forms of decomposition, process-oriented (Algorithmic) and object-oriented decomposition.

Algorithmic (Process Oriented) Decomposition: In Algorithmic decomposition, each module in the system denotes a major step in some overall process.



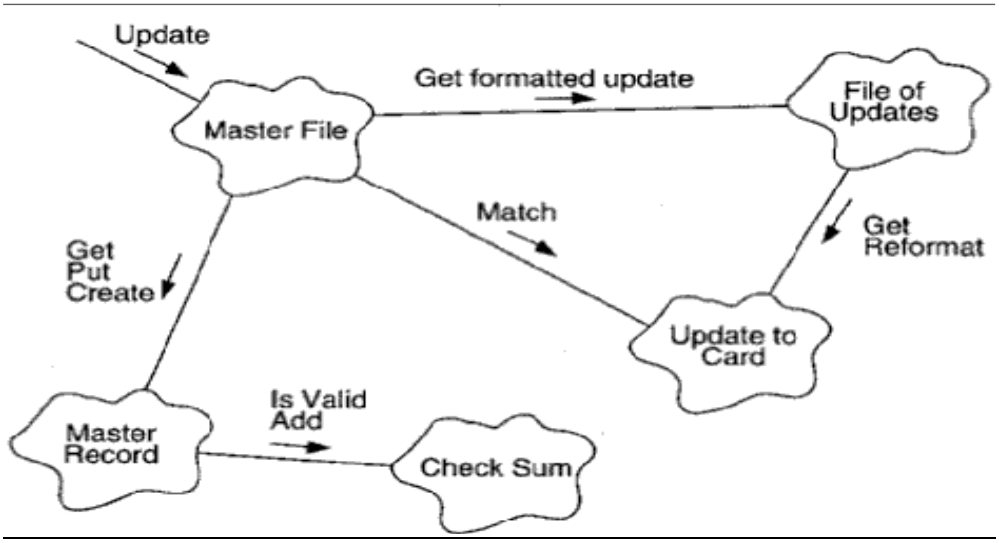
Algorithmic decomposition

Object oriented decomposition: Objects are identified as Master file and check sum which derive directly from the vocabulary of the problem.

We know the world as a set of autonomous agents that collaborate to perform some higher level behavior. Get formatted update thus does not exist as an independent algorithm; rather it is an operation associated with the object file of updates.

Calling this operation creates another object, update to card. In this manner, each object in our solution embodies its own unique behavior.

Each hierarchy in layered with the more abstract classes and objects built upon more primitive ones especially among the parts of the object structure, object in the real world. Here decomposition is based on objects and not algorithms.



Object Oriented decomposition

Algorithmic versus object oriented decomposition:

The algorithmic view highlighst 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.

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:

Definition: 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 role of the engineer as artist is particularly challenging when the task is to design an entirely new system.

The meaning of Design: In every engineering discipline, design encompasses the discipline approach we use to invent a solution for some problem, thus providing a path from requirements to implementation.

The purpose of design is to construct a system that.

1. Satisfies a given (perhaps) informal functional specification
2. Conforms to limitations of the 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.

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 buildings of models have a broad acceptance among all engineering disciplines largely because model building appeals to the principles of decomposition, abstraction and hierarchy.

Each model within a design describes a specific aspect of the system under consideration. Models give us the opportunity to fail under controlled conditions.

The Elements of Software design Methods:

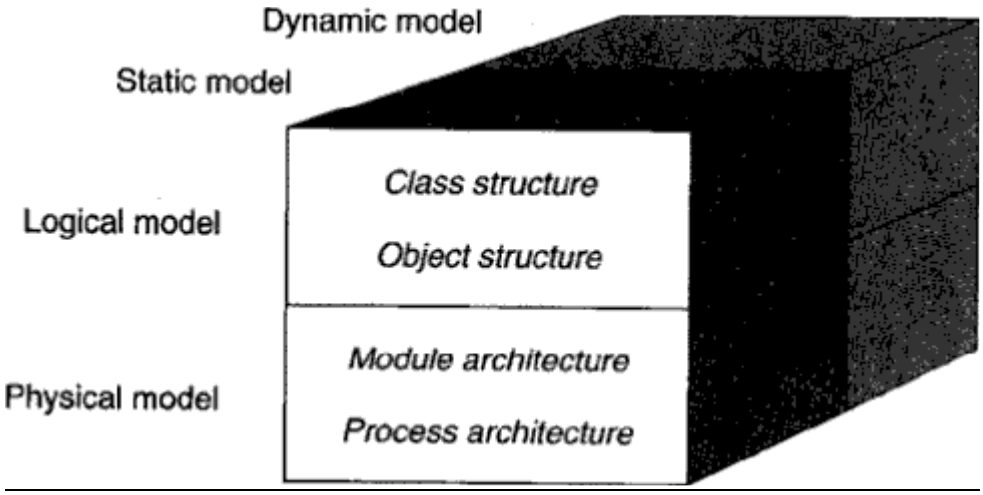
Design of complex software system involves an incremental and iterative process. Each method includes the following:

1. Notation: The language for expressing each model.
2. Process: The activities leading to the orderly construction of the system's mode.
3. Tools: The artifacts that eliminate the medium of model building and enforce rules about the models themselves, so that errors and inconsistencies can be exposed.

The models of Object Oriented Development:

The models of object oriented analysis and design reflect the importance of explicitly capturing both the class and object hierarchies of the system under design.

These models also over the spectrum of the important design decisions that we must consider in developing a complex system and so encourage us to craft implementations that embody the five attributes of well formed complex systems.



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.

Evolution of 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)

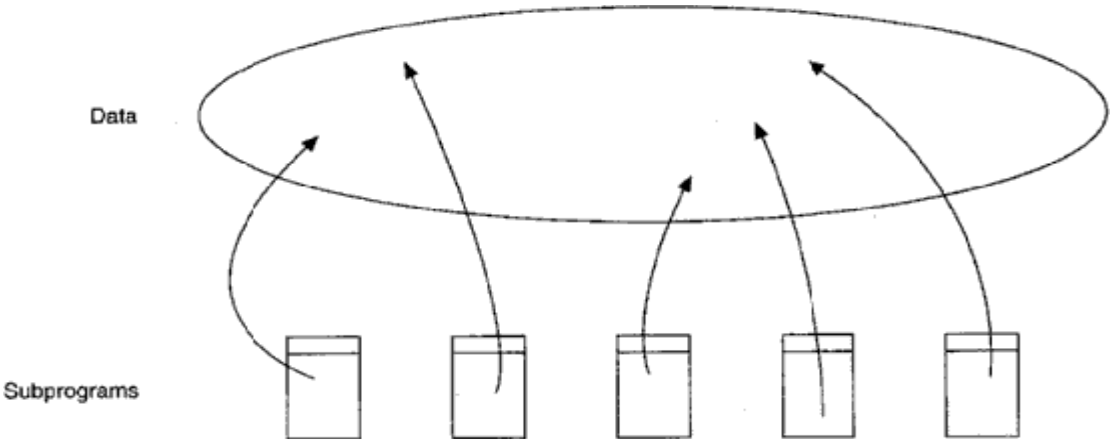
- Smalltalk 80 – Pure object oriented language
- C++ - Derived from C and Simula
- Ada83 – Strong typing; heavy Pascal influence
- Eiffel - Derived from Ada and Simula

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

<u>Topology of first and early second generation programming languages:</u>

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.

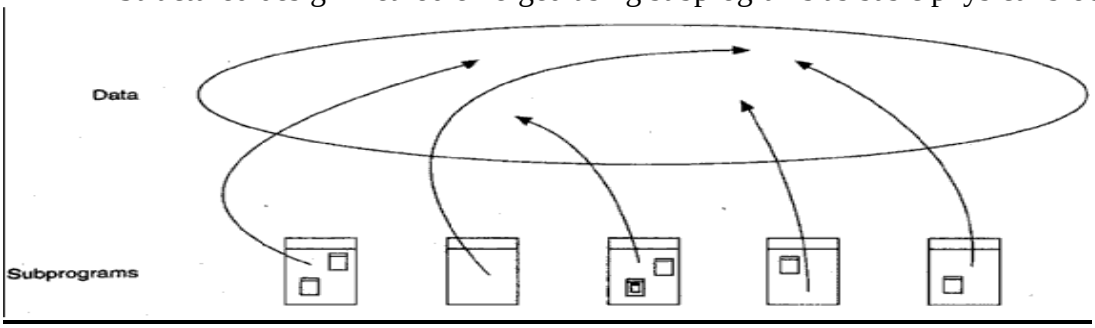


The Topology of First- and Early Second-Generation Programming Languages

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.

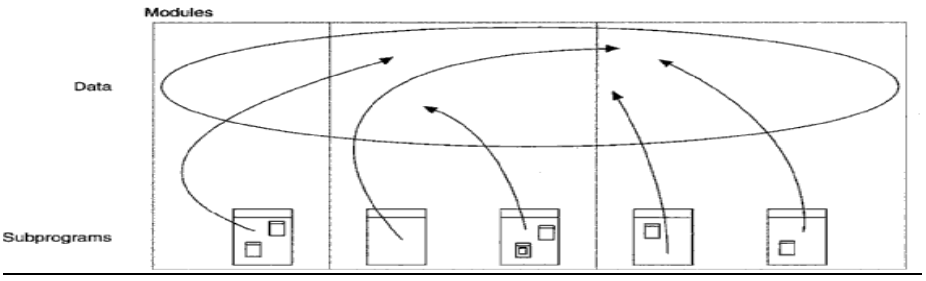


The Topology of Late Second- and Early Third-Generation Programming Languages

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.

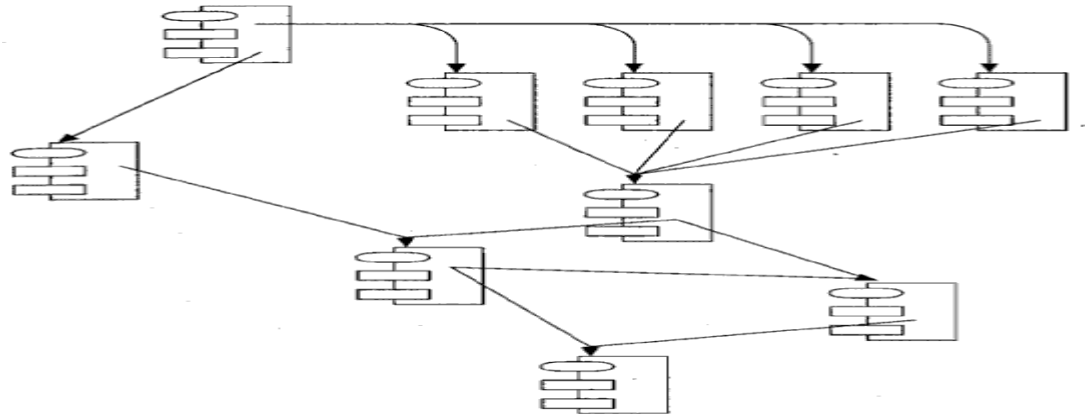


The Topology of Late Third-Generation Programming Languages

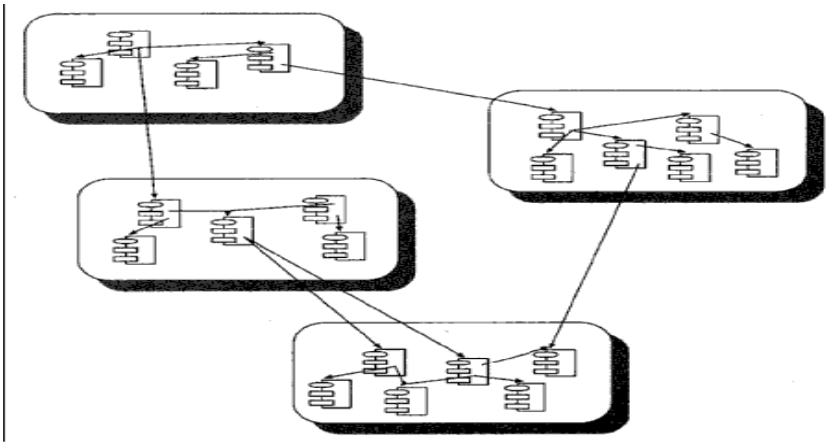
Topology of object and object oriented programming language :

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.



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



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 .
- 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

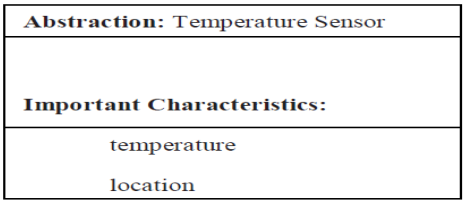
Abstraction :

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.

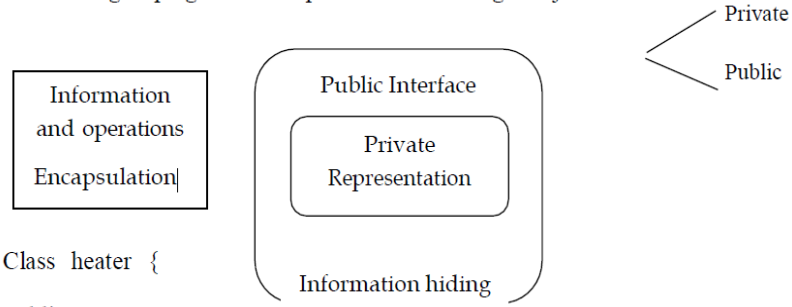
- **Entity abstraction:** An object that represents a useful model of a problem domain or solution domain entity.
- **Action abstraction:** An object that provides a generalized set of operations all of which program the same kind of function.
- **Virtual machine abstractions:** An object that groups 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 of a Temperature Sensor

Encapsulation

The act of grouping data and operations into a single object.




```
Class  heater  {
Public:
heater (location):
 ~ heater ( ):
void turnon ( );
void turnoff ( );
Boolean ison () const
private:
```

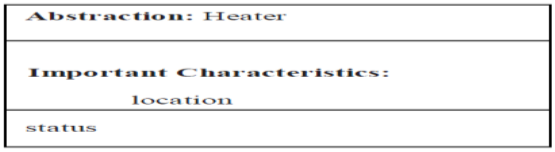


Figure 2.7: Abstraction of a Heater

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.

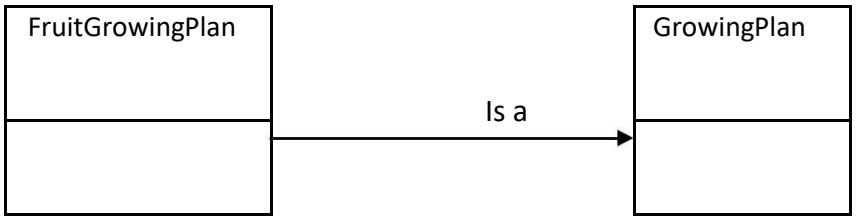


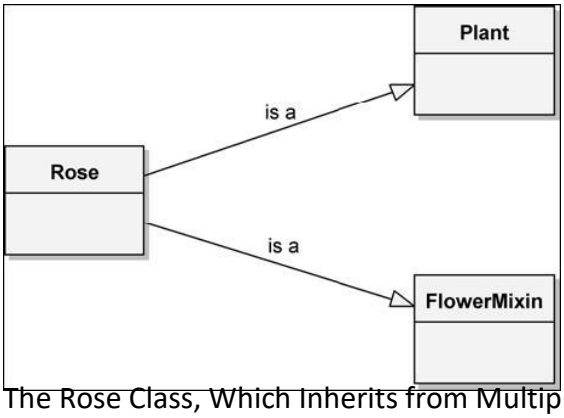
Fig 2.8: Class having one super class (Single Inheritance)

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 GrowingPlan, and VegetableGrowingPlan “is a” kind of GrowingPlan. Here, GrowingPlan is the more general superclass, and the others are specialized subclasses.

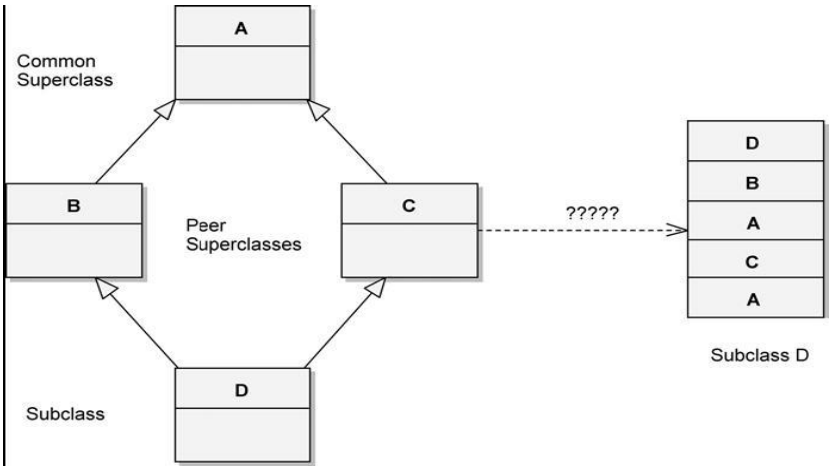
As we evolve our inheritance hierarchy, the structure and behavior that are common for different classes will tend to migrate to common super classes. This is why we often speak of inheritance as being a generalization/specialization hierarchy.

Super classes represent generalized abstractions, and subclasses represent specializations in which fields and methods from the super class are added, modified, or even hidden.

Examples of Hierarchy: Multiple Inheritance



The Rose Class, Which Inherits from Multiple Superclasses (Multiple Inheritance)



The Repeated Inheritance

Repeated inheritance occurs when two or more peer superclasses share a common superclass.

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 GrowingPlan 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 GrowingPlan; when we destroy the Garden object, we in turn destroy the GrowingPlan 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 superclass. 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 ActiveTemperatureSensor, 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 setpoint. 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 i.e. 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 exists between various versions of the 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.

Application of Object Model:

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.