UNIT-1

Object:

An object is a real-world element in an object–oriented environment that may have a physical or a conceptual existence. Objects can be modelled according to the needs of the application. An object may have a physical existence, like a customer, a car, etc.; or an intangible conceptual existence, like a project, a process, etc.

Object Oriented Analysis and Design (OOAD):

Object-oriented analysis and design (OOAD) is a popular technical approach for analyzing and designing an application, system, or business by applying object-oriented paradigm, as well as using visual modeling throughout the development life cycle for better communication and product quality.

Object Oriented Analysis (OOA):

The main difference between Object-Oriented Analysis and other forms of analysis is that by the object-oriented approach we organize requirements around objects, which integrate both behaviors (processes) and states (data) modeled after real world objects that the system interacts with.

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

- Find the objects
- Organize the objects
- Describe how the objects interact
- Define the behavior of the objects
- Define the internals of the objects

Object Oriented Design (OOD):

Object Oriented Design involves implementation of the conceptual model produced during Object Oriented Analysis. OOD concepts in the analysis model which are technology independent are mapped onto implementing classes, Constraints are identified and Interfaces are designed resulting in a model for the solution domain.

Object Oriented Paradigm:

Object oriented paradigm based upon objects (having data in the form of fields often known as attributes and code in the form of procedures often known as methods) that aims to incorporate the advantages of modularity and reusability.

1.1 The Structure of Complex Systems:

Systems with a set of parts are called as elements and a set of connections between these parts are called relations. These parts can be ordered or unordered.

E.g.: Parts of a car, Organs in our body.

Software might be referred to as simple or complex depending upon their functionality and behavior. Generally, industrial-strength software are more complex than those developed individually or by user developers.

Time and space are considered to be general complexities. Additionally, maintaining integrity of hundreds of thousands of records while allowing concurrent updates and queries; or managing command and control of real-world entities like air traffic are also examples of complex systems.

Structure of a Personal Computer:

A device of moderate complexity, which has been evolved from small working sub systems like gate architecture, for instance.

- Composed of the same major elements – CPU, monitor, keyboard, secondary storage
- An element can be taken and decomposed further and can be studied separately.
- The collaborative activity of each major part (hierarchically arranged) causes a computer system to function.
- The hierarchy of the computer system also represents different levels of abstraction – each built upon another and in each level of abstraction, a collection of devices are found to be collaborating one another to provide functions to higher layers.

Structure of a Plants and Animals:

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.

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.

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.

1.2 The inherently Complexity of Software:

The complexity of software is an essential property not an accidental one. The inherent complexity derives from four elements.

a. The complexity of the problem domain

- Complex requirements
- Decay of system

Often, although software developers have the knowledge and skills required to develop software they usually lack detailed knowledge of the application domain of systems. This affects their ability to understand and express accurately the requirements for the system to be built which come from the particular domain. Note, that these requirements are usually themselves subject to change.

External complexity usually springs from the difference of thoughts that exists between the users of a system and its developers. Users may have only vague ideas of what they want in a software system.

Users and developers have different perspectives on the nature of the problem and make different assumptions regarding the nature of the system. A further complication is that the requirement of a software system is often change during its development.

It is software maintenance when we correct errors, evolution when we respond to changing environments and preservations, when we continue to use extraordinary means to keep an ancient and decaying piece of software in operation.

b. 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 intensive systems cannot be developed by single individuals. They require teams of developers.

Developer must strive to write the less code by using the clever and powerful techniques. Even if we decompose our implementation in meaningful ways, we still end up with hundreds and sometimes even thousand modules.

No single person can totally understand the complexity of system. There are always significant challenges associated with team development more developers' means more complex communication and hence more difficult coordination.

c. 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.

The third reason is the danger of flexibility. Flexibility leads to an attitude where developers develop system components themselves rather than purchasing them from somewhere else. The software development is different: most of the software companies develop every single component from scratch. Construction industry has standards for quality of row materials, few such standards exist in the software industry.

d. The problem of characterizing the behavior of discrete systems

- Numerous possible states
- Difficult to express all states

The final reason for complexity according to Booch is related to the difficulty in describing the behavior of software systems. Humans are capable of describing the static structure and properties of complex systems if they are properly decomposed, but have problems in describing their behavior. This is because to describe behavior, it is not sufficient to list the properties of the system. It is also necessary to describe the sequence of the values that these properties take over time.

Within a large application, there may be hundreds or even thousands of variables well as more than one thread of control. The entire collection of these variables as well as their current values and the current address within the system constitute the present state of the system with discrete states. Discrete systems by their very nature have a finite numbers of possible states.

e. The Consequences of Unrestrained Complexity

The more complex the system, the more open it is to total breakdown. Rarely would a builder think about adding a new sub-basement to an existing 100-story building; to do so would be very costly and would undoubtedly invite failure. Amazingly, users of software systems rarely think twice about asking for equivalent changes. Besides, they argue, it is only a simple matter of programming.

1.3 Attributes of a complex system:

There are five attribute common to all complex systems. They are as follows:

1. Hierarchical and interacting subsystems

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

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. 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. 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 and animals, or of larger structures, such as vascular systems, also found in both plants and animals.

5. 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.

Very often subsystems are decomposed again into subsystems, which are decomposed and so on. The way how this decomposition is done and when it is stopped, i.e. which components are considered primitive, is rather arbitrary and subject to the architects decision.

The decomposition should be chosen, such that most of the coupling is between components that lie in the same subsystem and only a loose coupling exists between components of different subsystem. This is partly motivated by the fact that often different individuals are in charge with the creation and maintenance of subsystems and every additional link to other subsystems does imply an higher communication and coordination overhead.

Certain design patterns re-appear in every single subsystem. Examples are patterns for iterating over collections of elements, or patterns for the creation of object instances and the like.

The development of the complete system should be done in slices so that there is an increasing number of subsystems that work together. This facilitates the provision of feedback about the overall architecture.

1.4 Organized and Disorganized Complexity

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.

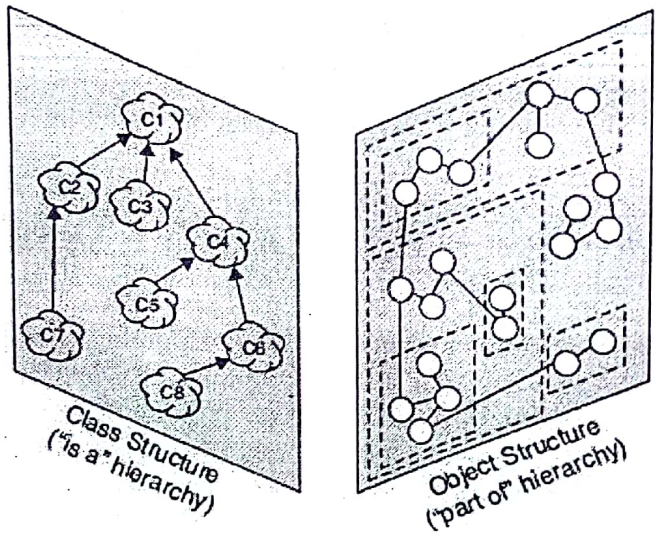


Fig 1.4.1 The Key Hierarchies of Complex

Systems **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, we find that

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

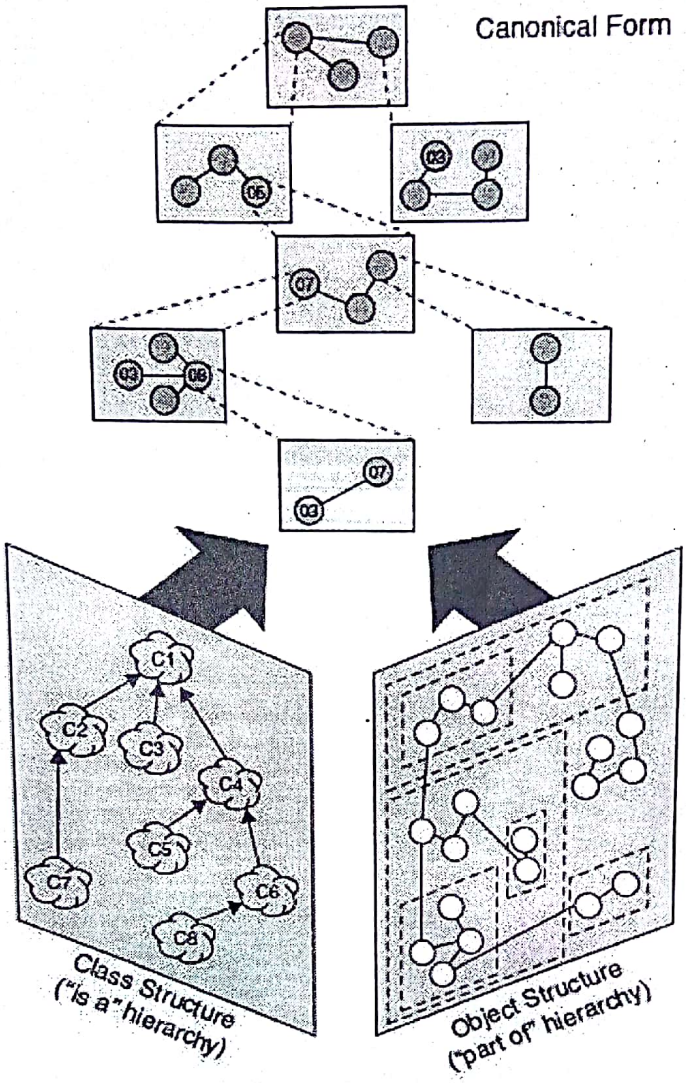


Figure 1.4: Canonical form of a complex system

The figure 1.4 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.

Approaching a Solution

The Limitations of the human capacity for dealing with complexity:

- Dealing with complexities
- Memory
- Communications

When we devise a methodology for the analysis and design of complex systems, we need to bear in mind the limitations of human beings, who will be the main acting agents, especially during early phases. Unlike computers, human beings are rather limited in dealing with complex problems and any method need to bear that in mind and give as much support as possible.

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.

The analysis process is a communication intensive process where the analyst has to have intensive communications with the stakeholders who hold the domain knowledge. Also the design process is a communication intensive process, since the different agents involved in the design need to agree on decompositions of the system into different hierarchies that are consistent with each other.

1.5 Bringing Order to chaos

Principles that will provide basis for development

1. **The Role of Abstraction:** Abstraction is an exceptionally powerful technique for dealing with complexity. Unable to master the entirely of a complex object, we chose 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.

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.

2. **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.

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 thereof. Similarly, an object that is up in the part-of hierarchy represents a rather coarse-grained and complex objects, assembled from a number of objects, while objects that are leafs are rather fine grained. But note that there are many other forms of patterns which are nonhierarchical: interactions, 'relationships'.

3. **The role of 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.

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

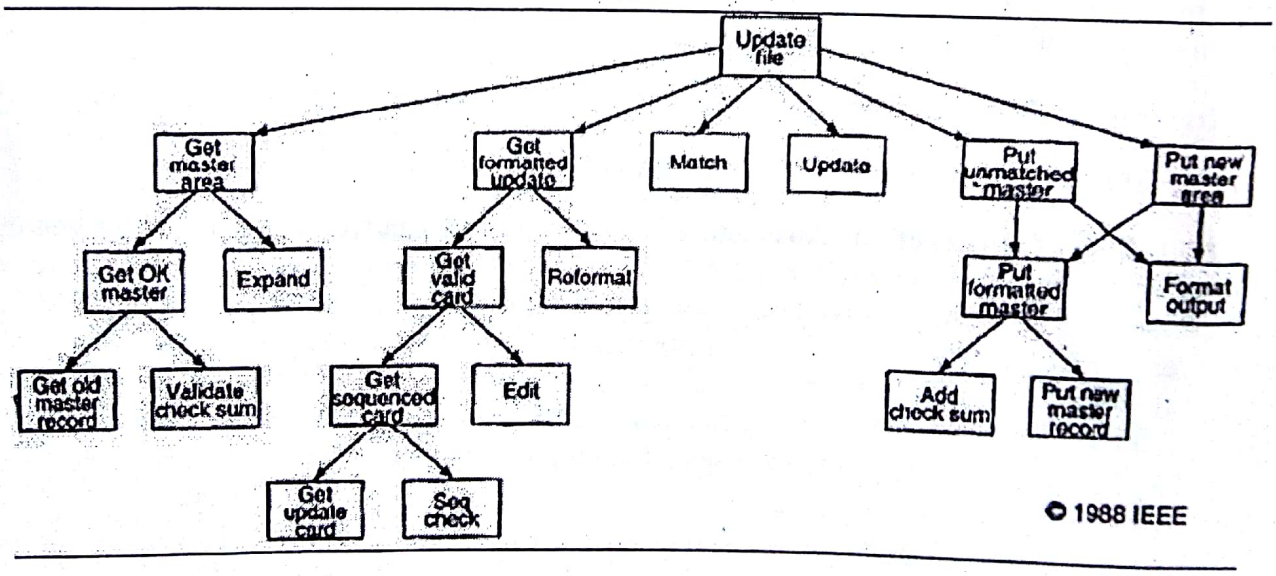


Figure: Algorithmic decomposition

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

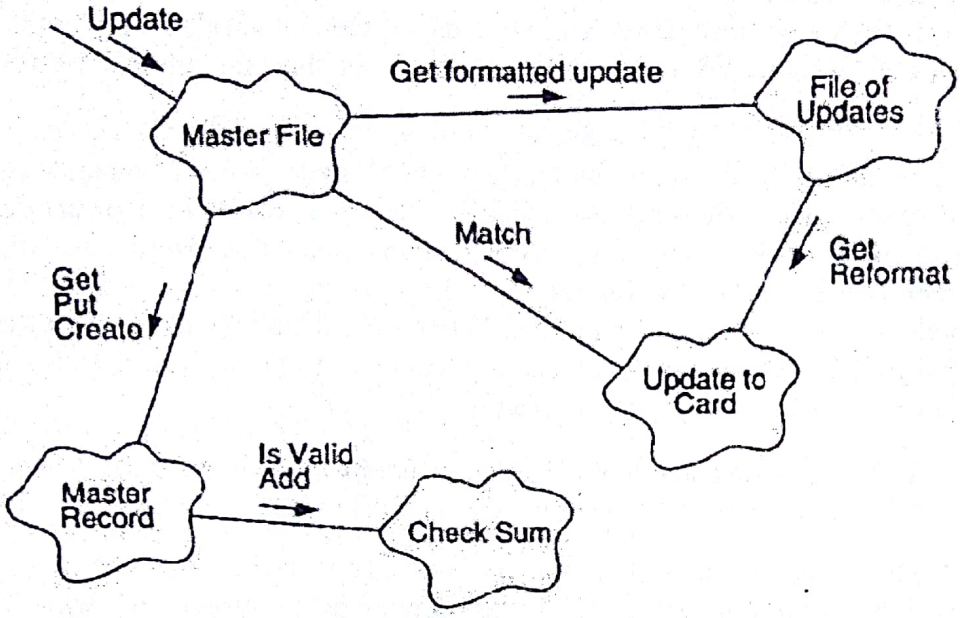


Figure 1.3: Object oriented decomposition

c. **Algorithmic versus object oriented decomposition:** 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.

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.

1.6 On Designing Complex Systems

Engineering as a Science and an Art: 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.

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 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. We evaluate each model under both expected and unusual situations and then after them when they fail to behave as we expect or desire. More than one kind of model is used on order to express all the subtleties of a complex system.

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.

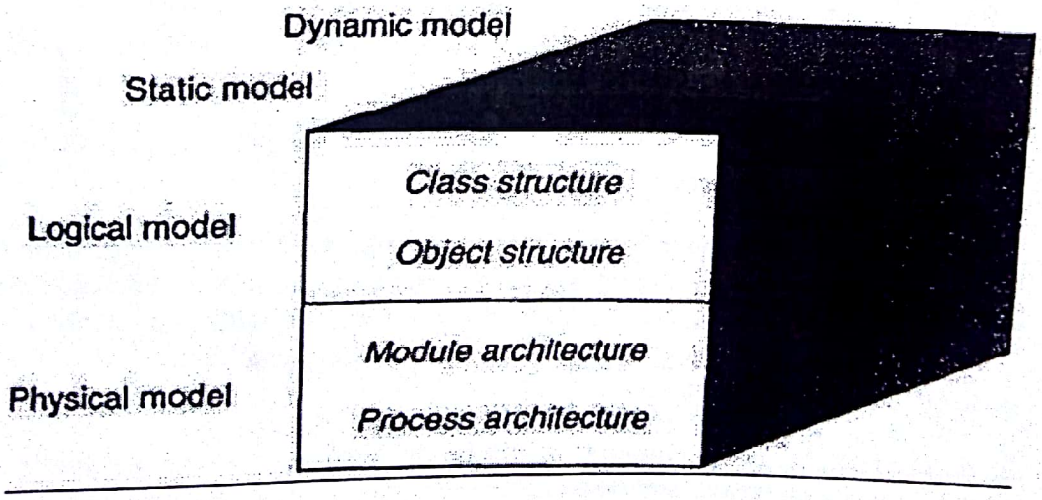


Figure 1.4: Models of object oriented development

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.

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.

1.7 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.

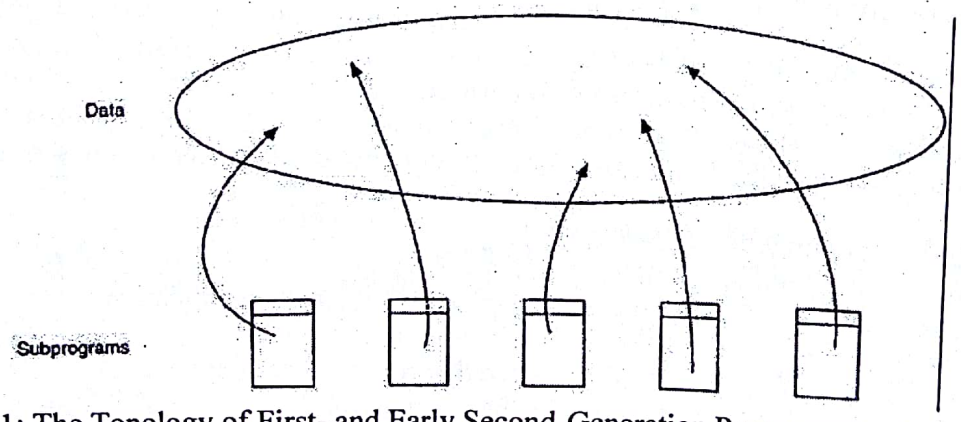


Fig 1.7.1: The Topology of First- and Early Second-Generation Programming Languages

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.

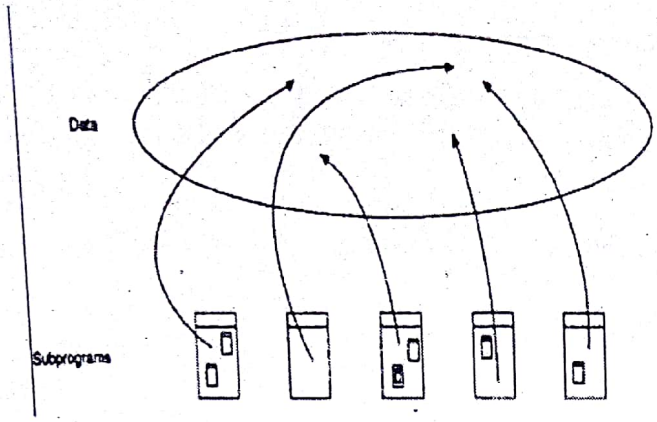


Fig 1.7.2: The Topology of Late Third-Generation Programming Languages

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.

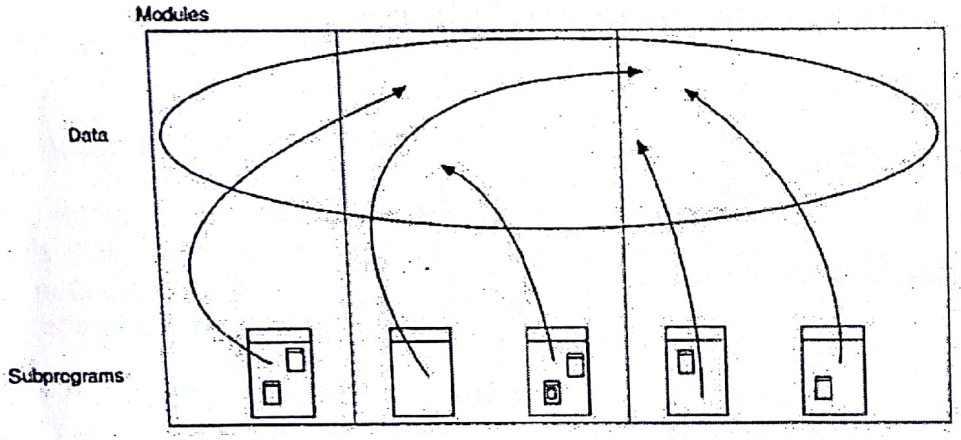


Fig 1.7.3: The Topology of Late Third-Generation Programming Languages

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

- Data driven design method emerged for data abstraction.
- 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.

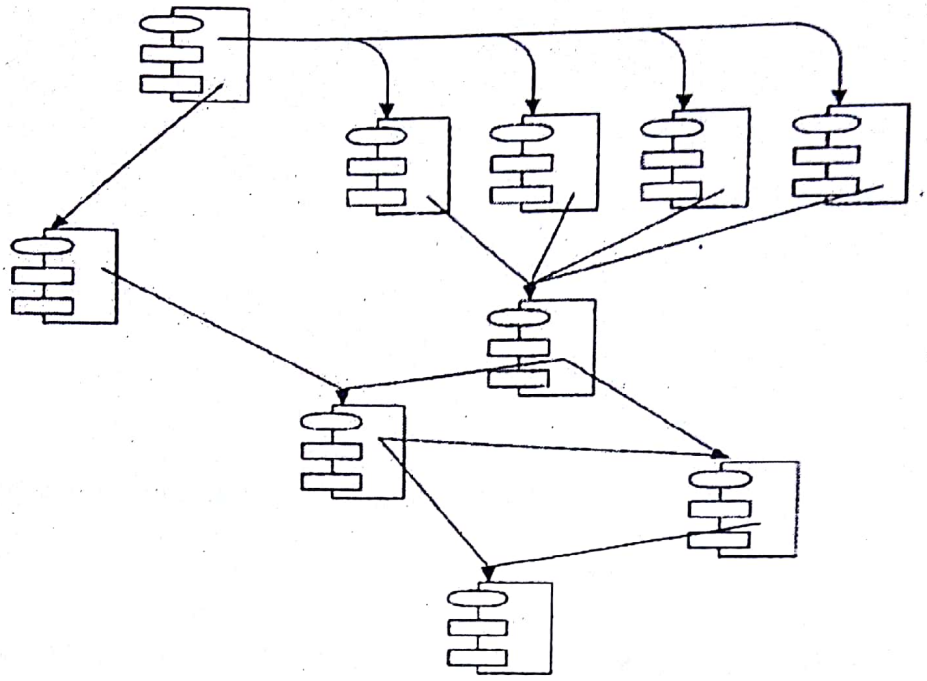


Fig 1.7.4: The Topology of Small- to Moderate-Sized Applications Using Object-Based and Object-Oriented Programming Languages

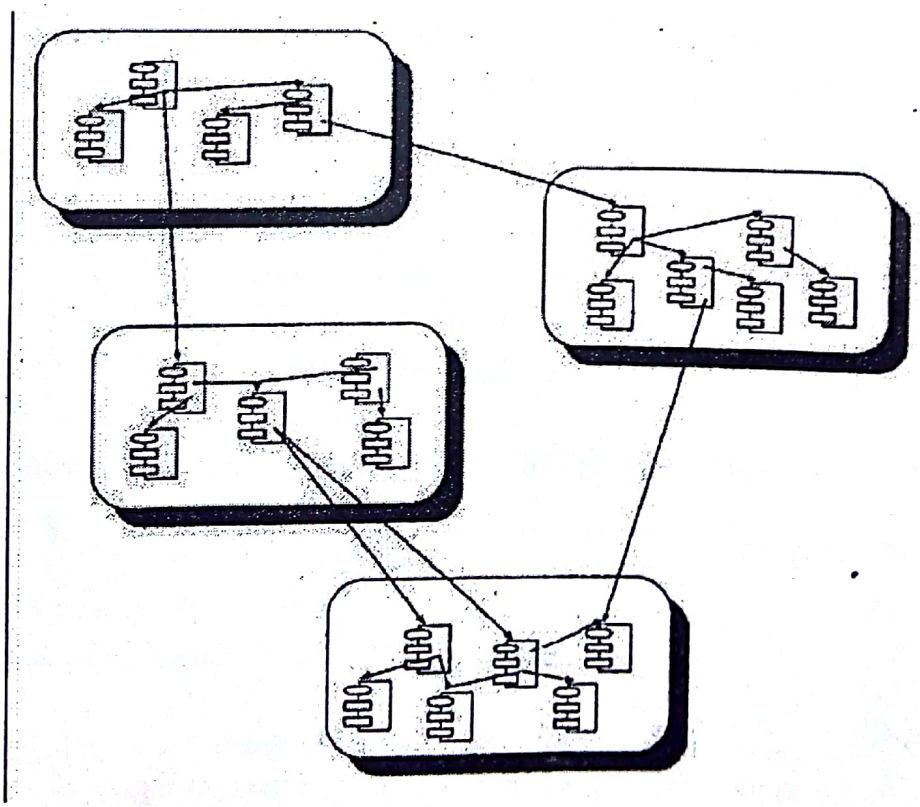


Fig 1.7.5: The Topology of Large Applications Using Object-Based and Object-Oriented Programming Languages

1.8 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 objects 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.

1.9 Elements of Object Model

Kinds of Programming Paradigms: 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.

- 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.

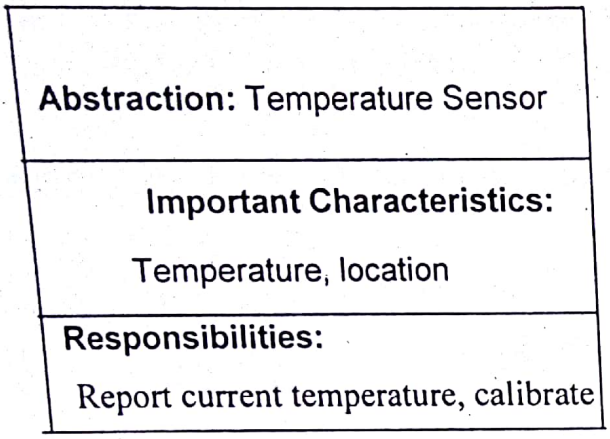
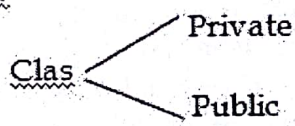
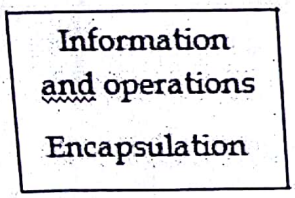


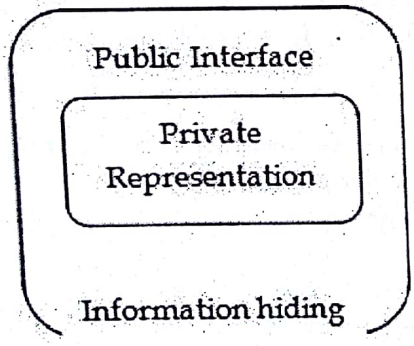
Figure 1.9.1: Abstraction of a Temperature Sensor

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:
```

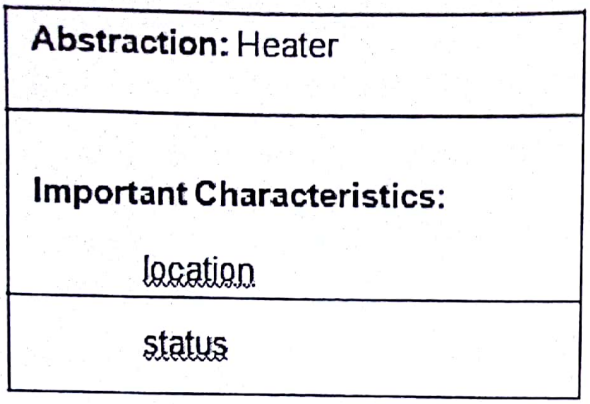


Figure 2.7: Abstraction of a Heater

Modularity

The act of partitioning a program into individual components is called modularity. It is reusable component which reduces complexity to some degree. 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.

- modules can be compiled separately. modules in C++ are nothing more than separately compiled files, generally called header files.
- Modules are units in pascal and package body specification in Ada.

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.

Examples of Hierarchy: Multiple Inheritance

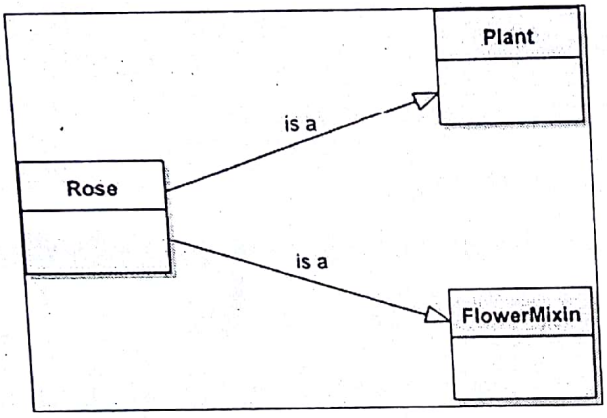


Figure 1.9.2: The Rose Class, Which Inherits from Multiple Super classes (Multiple Inheritance)

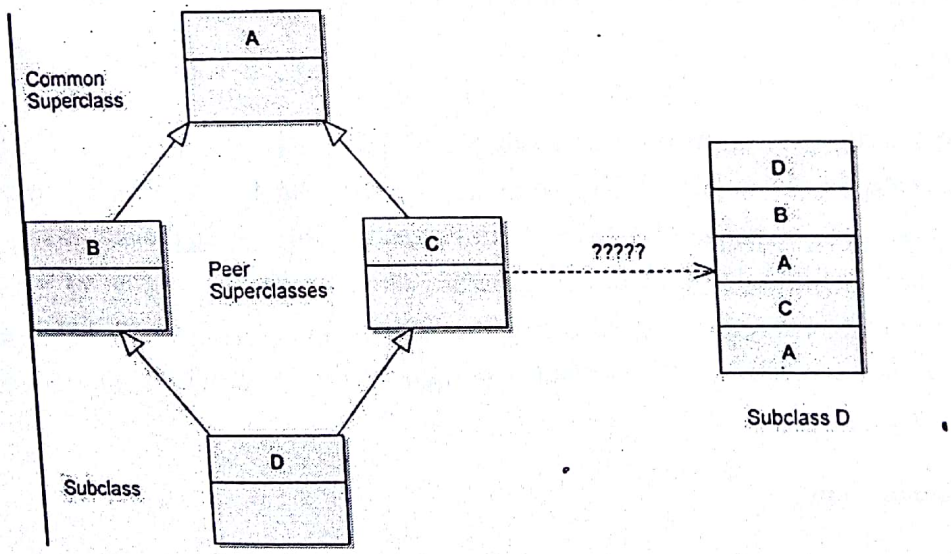


Figure 1.9.3: The Repeated Inheritance

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

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. 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.

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).

Concurrency

Concurrency focuses upon process abstraction and synchronization. Each object may represent a separate thread of actual (a process abstraction). Such objects are called 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.

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.

- Data that exists between executions of a program
- Data that exists between various versions of the program
- Data that outlines the Program.

1.10 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.