

# 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.

---

**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.

**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.

## —→ The structure of Complex Systems

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

**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.

**The structure of Plants:** Plants are complex multicellular organism which are composed of cells which in 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.

All parts at the same level of abstraction interact in well-defined ways. For example, at the highest level of abstraction, roots are responsible for observing water and minerals from the soil. Roots interact with stems, which transport these raw materials up to leaves. The leaves in turn use the water and minerals provided by the stems to produce food through photosynthesis.

**The structure of Animals:** Animals exhibit a multicellular 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.

**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.

## —→ **The Inherent Complexity of Software**

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

1. The complexity of the problem domain.
2. The difficulty of managing the developmental process.
3. The flexibility possible through software .
4. The problems of characterizing the behavior of discrete systems.

### **1. The complexity of the problem domain**

The first reason has to do with the relationship between the application domains for which software systems are being constructed and the people who develop them. Often, although software developers have the knowledge and skills required to develop software they usually lack detailed knowledge of the application domain of such 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. They evolve during the construction of the system as well as after its delivery and thereby they impose a need for a continuous evolution of the system. Complexity is often increased as a result of trying to preserve the investments made in legacy applications. In such cases, the components which address new requirements have to be integrated with existing legacy applications. This results into interoperability problems caused by the heterogeneity of the different components which introduce new complexities.

### **2. The Difficulty of Managing the Development Process**

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. This adds extra overhead to the process since the developers have to communicate with each other about the intermediate artifacts they produce and make them interoperable with each other. This complexity often gets even more difficult to handle if the teams do not work in one location but are geographically dispersed. In such situations, the management of these processes becomes an important subtask on its own and they need to be kept as simple as possible. No person can understand the system whose size is measured in hundreds of thousands, or even millions of lines of code. Even if we decompose our implementation in meaningful ways, we still end up with hundreds and sometimes even thousand modules. The amount of work demands that we use a team of developers and there are always significant challenges associated with team development more developers means more complex communication and hence more difficult coordination.

### **3. The flexibility possible through software**

Software is flexible and expressive and thus encourages highly demanding requirements, which in turn lead to complex implementations which are difficult to assess. 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. Unlike other industrial sectors, the production depth of the software industry is very large. The construction or automobile industries largely rely on highly specialized suppliers providing parts. The developers of these industries just produce the design, the part specifications and assemble the parts delivered.

The software development is different: most of the software companies develop every single component from scratch. Flexibility also triggers more demanding requirements which make products even more complicated.

Software offers the ultimate flexibility. It is highly unusual for a construction firm to build an on site steel mill to forge (create with hammer) custom girders (beams) for a new building. Construction industry has standards for quality of raw materials, few such standards exist in the software industry.

#### **4.The problem of characterizing the behavior of discrete systems**

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.

### → **The five Attributes of a complex system:**

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

#### **1.Hierarchic 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.

The fact that many complex systems have a nearly decomposable, and hierarchic structure is a major facilitating factor enabling us to understand, describe, and even "see" such systems their parts.

#### **2.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.Separation of Concerns**

Hierarchic systems decomposable because they can be divided into identifiable parts; he calls them nearly decomposable because their parts are not completely independent. This leads to another attribute common to all complex systems:

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 or 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.

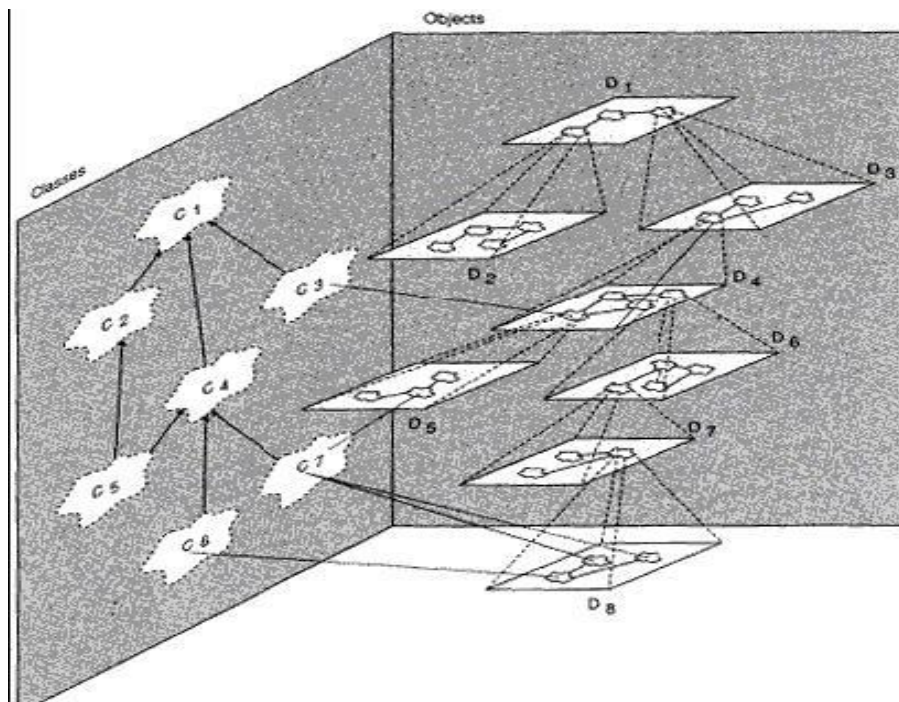
## → Organized and Disorganized Complexity

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, we find that 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 : Canonical form of a complex system*

The figure 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 than there are classes. The point in identifying classes is therefore to have a vehicle to describe only once all properties that all instances of the class have.

### **The Limitations of the human capacity for dealing with complexity:**

Object model is the organized complexity of software. As we begin to analyze a complex software system, we find many parts that must interact in a multitude of intricate ways with little commonality among either the parts or their interactions. This is an example of disorganized complexity. In complex system, we find many parts that must interact in a multitude of intricate ways with little commonality among either the parts or their intricate. This is an example in an air traffic control system, we must deal with states of different aircraft at once, and involving such it is absolutely impossible for a single person to keep track of all these details at once.

### **→ Bringing Order to chaos**

Principles that will provide basis for development

- Abstraction
- Hierarchy
- Decomposition

**1. The Role of Abstraction:** Abstraction is an exceptionally powerful technique for dealing with complexity. Unable to master the entirety 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. 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 coping 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.

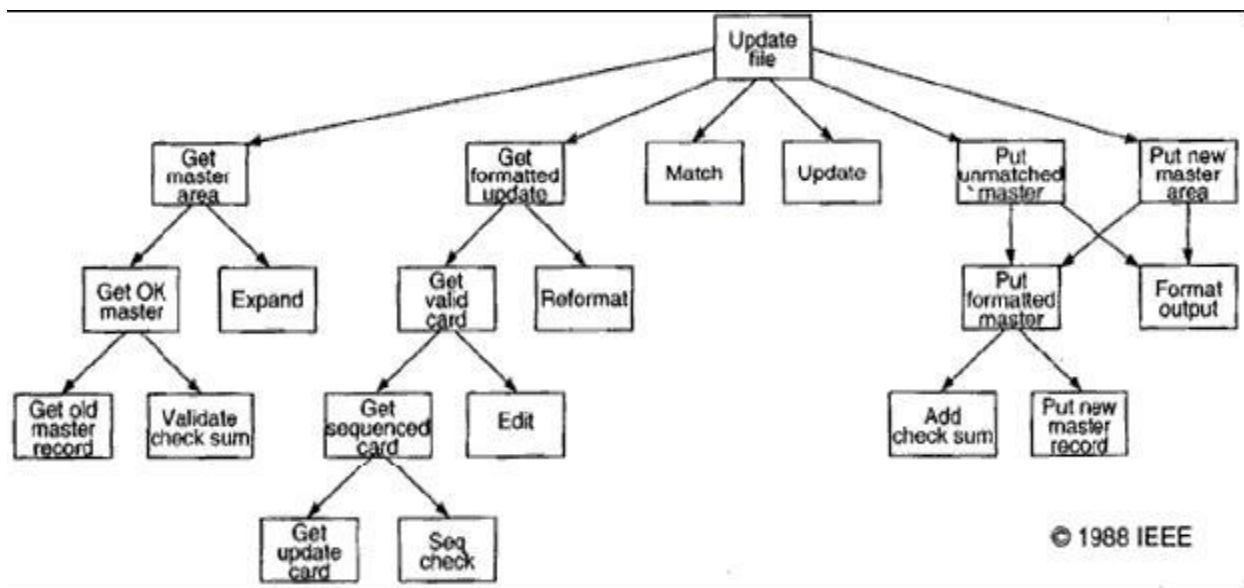
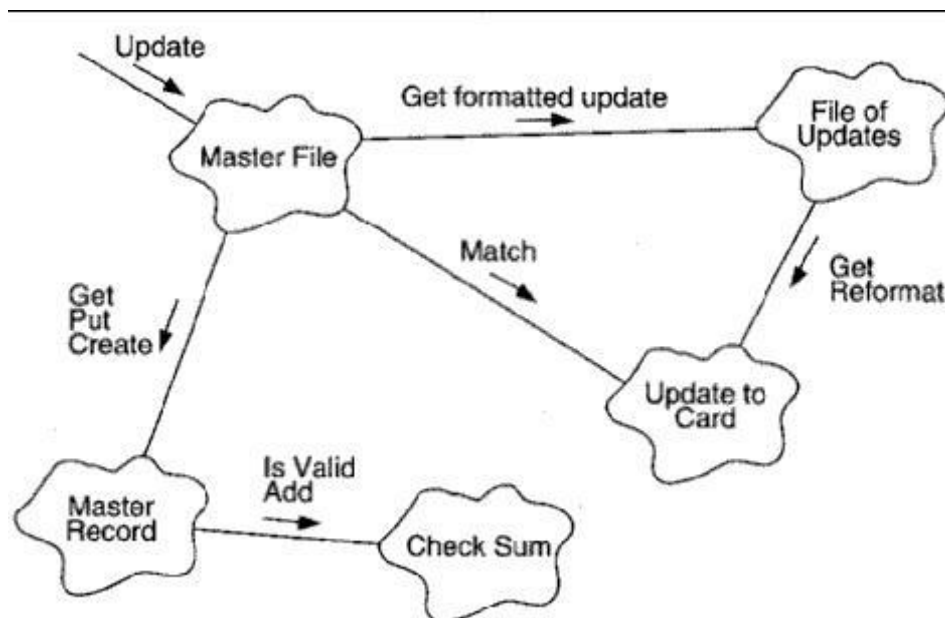


Figure : Algorithmic decomposition

**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 is 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*

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

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

objects that encapsulate both a state and a certain behavior. Then communication among these objects leads to the desired solutions.

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

## —→ **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 fordoing 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.

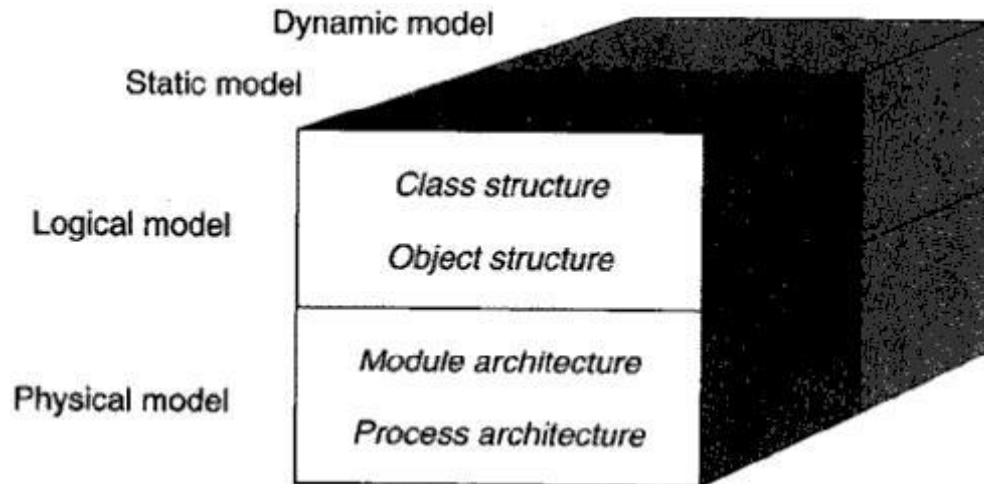
**The Importance of Model Building:** The buildings of models have a broad acceptance among all engineering disciplines largely because model building appeals to the principles of decomposition, abstraction and hierarchy. Each model within a design describes a specific aspect of the system under consideration. 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 cover 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 :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 compilable modules and the process architecture identifies how objects are distributed at run-time over different operating system processes and identifies the relationships between those. Again for this physical model a static perspective is defined that considers the structure of module and process architecture and a dynamic perspective identifies process and object activation strategies and inter-process communication. Object-orientation has not, however, emerged fully formed. In fact it has developed over a long period, and continues to change.

## **PART-II**

### **The Object Model**

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

#### **→ The Evolution of the object Model**

##### **The generation of programming languages**

Wegner has classified some of more popular programming languages in generations according to the language features.

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.

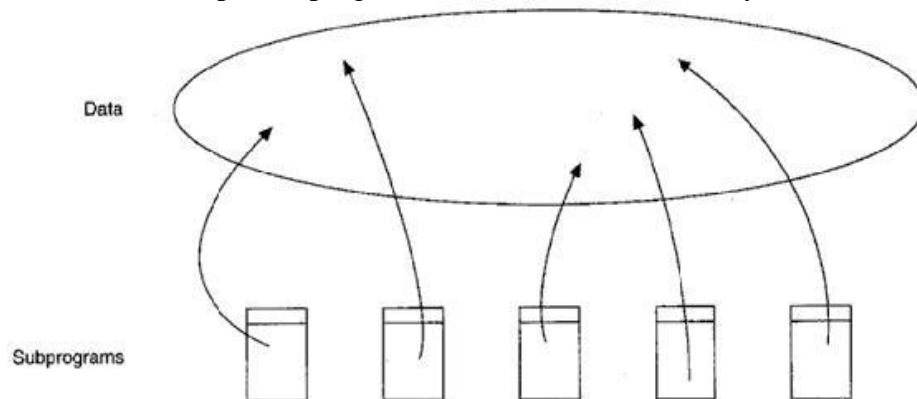


Fig : The Topology of First- and Early Second-Generation Programming Languages

### 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.

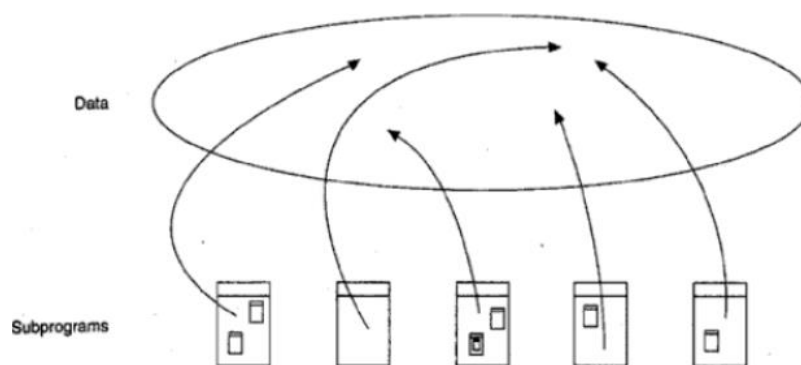


Fig : The Topology of Late Second- and Early Third-Generation Programming Languages

### 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. compiled module.
- Support modular structure.

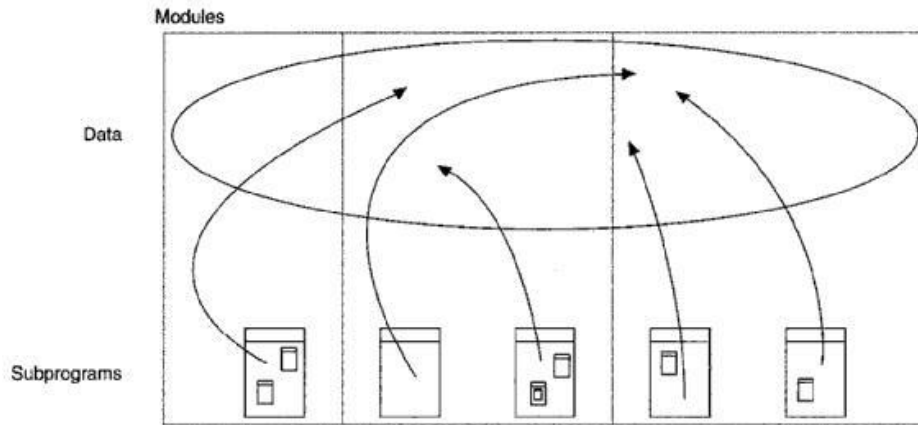


Fig : The Topology of Late Third-Generation Programming Languages

### Topology of object and object oriented programming language

Two methods for complexity of problems

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

(ii) Theories regarding the concept of a type appeared

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

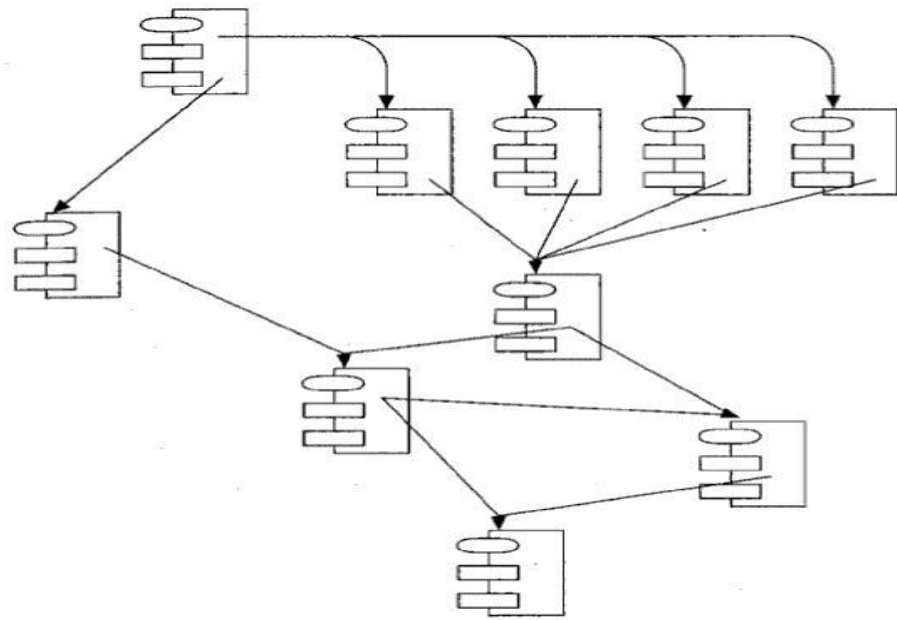


Fig : The Topology of Small- to Moderate-Sized Applications Using Object-Based and Object-Oriented Programming Languages

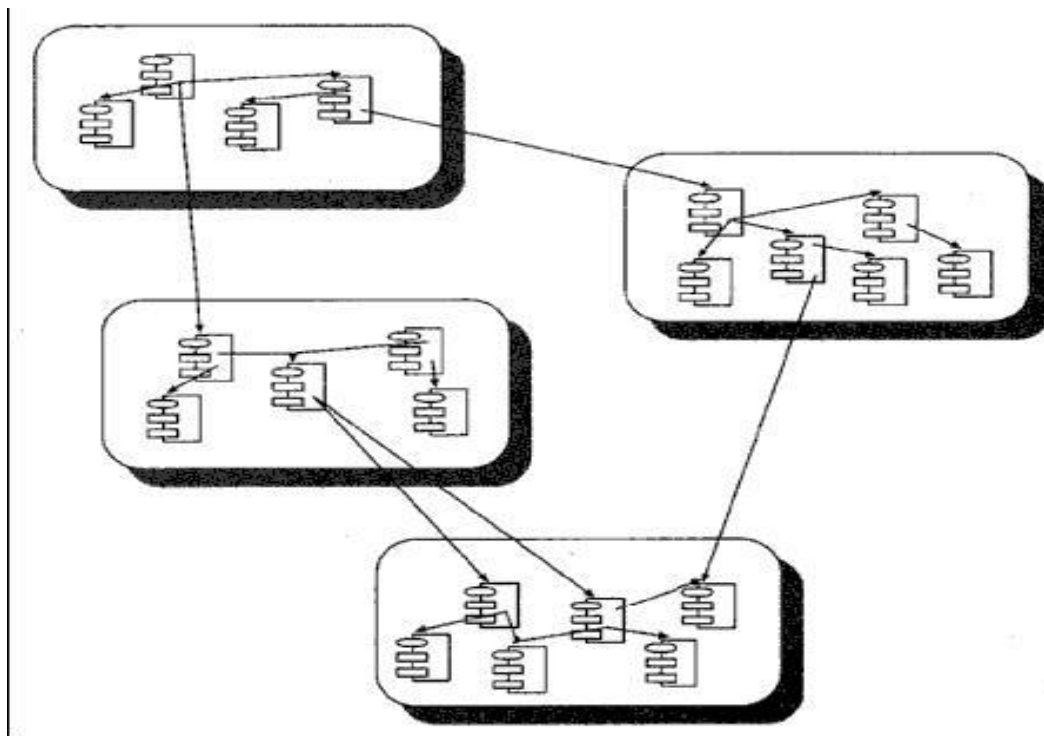


Fig : The Topology of Large Applications Using Object-Based and Object-Oriented Programming Languages.

## → **Foundations of the object model**

In structured design method, build complex system using algorithm as their fundamental building block. An object oriented programming language, class and object as basic building block.

Following events have contributed to the evolution of object-oriented concepts:

- Advances in computer architecture, including capability systems and hardware support for operating systems concepts
- Advances in programming languages, as demonstrated in Simula, Smalltalk, CLU, and Ada.
- Advances in programming methodology, including modularization and information hiding. We would add to this list three more contributions to the foundation of the object model:
  - Advances in database models
  - Research in artificial intelligence
  - Advances in philosophy and cognitive science

### **OOA (Object Oriented analysis)**

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

### **OOD (Object oriented design)**

During user requirement phase, OOD involves understanding of the application domain and build an object model. Identify objects; it is methods of design showing process of object oriented decomposition. Object oriented design is a method of design encompassing the process of object oriented decomposition and a notation for depicting both logical and physical as well as static and dynamic models of the system under design.

### **OOP (Object oriented programming)**

During system implementation phase, it 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**

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

### **1. 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.

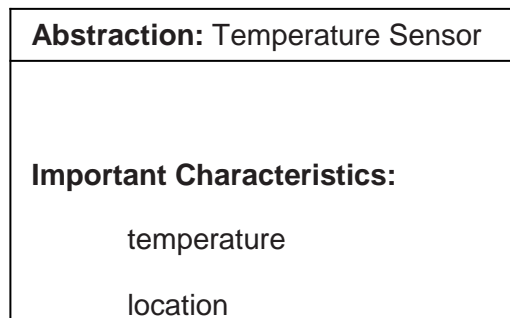


Figure 2.6: Abstraction of a Temperature Sensor

## 2. Encapsulation

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

## 3. Modularity

. 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.

## 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.

## 4. Hierarchy

Hierarchy is a ranking or ordering of abstractions. Encapsulation hides complexity inside new or 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 class shares structure or behaviors defined in one (single inheritance) 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.



Fig : Class having one superclass (Single Inheritance)

In this case, FruitGrowingPlan is more specialized, and GrowingPlan is more general. The same could be said for GrainGrowingPlan or VegetableGrowingPlan, that is, GrainGrowingPlan "is a" kind of GrowingPlan, and VegetableGrowingPlan "is a" kind of GrowingPlan. Here, GrowingPlan is the more general superclass, and the others are specialized subclasses.



## Examples of Hierarchy: Multiple Inheritance

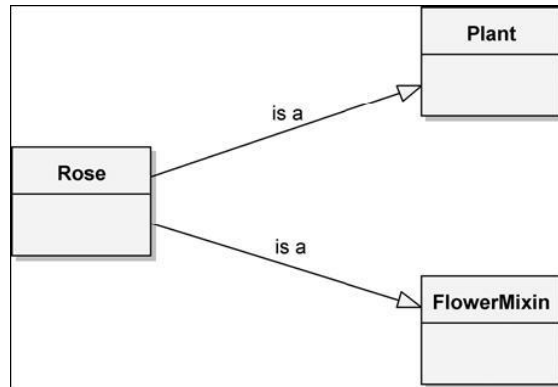


Figure 2.9: The Rose Class, Which Inherits from Multiple Superclasses (Multiple Inheritance)

### 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.

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

#### 1. 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 sense 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.

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).

## 2. 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.

## 3. 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.

## → **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 resilient 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.

## **Previous Questions**

1. Explain structure of complex systems
2. State and explain five attributes of a complex system.
3. Illustrate the design of complex system.
4. Discuss the evolution of object model with an example.
5. a. Explain the benefits of object model.  
b. Explain the application of an object model.

## **Two-marks Questions**

1. Define Software complexity.
2. List out five attributes of complex system
3. list elements of object model.
4. Distinguish between object-oriented and object-based languages.

## UNIT II

### Classes and Objects

**Classes and Objects:** Nature of Object, Relationships among objects, nature of a class, Relationship among classes, interplay of classes and objects, Identifying classes and objects, Importance of proper classification, Identifying classes and objects, Key abstractions and mechanisms

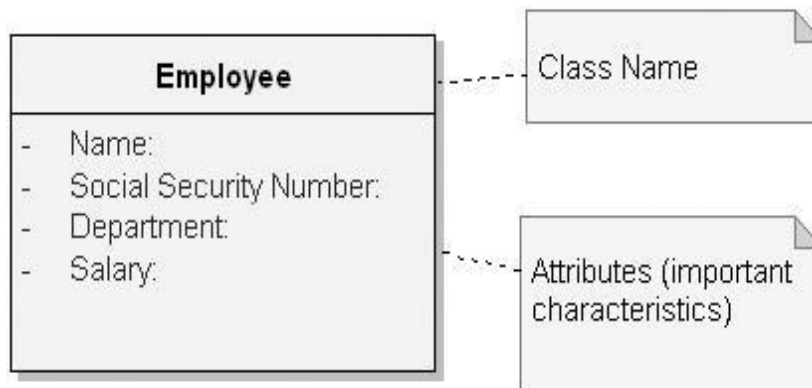
#### → The nature of an object

An object has state, behavior, and identity; the structure and behavior of similar objects are defined in their common class; the terms instance and object are interchangeable.

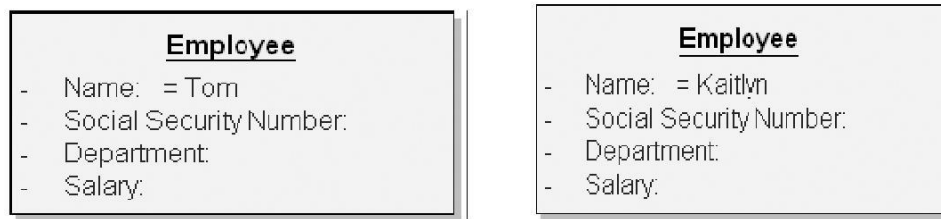
#### **State**

The state of an object encompasses all of the (usually static) properties of the object plus the current (usually dynamic) values of each of these properties.

Consider a vending machine that dispenses soft drinks. The usual behavior of such objects is that when someone puts money in a slot and pushes a button to make a selection, a drink emerges from the machine. What happens if a user first makes a selection and then puts money in the slot? Most vending machines just sit and do nothing because the user has violated the basic assumptions of their operation. Stated another way, the vending machine was in a state (of waiting for money) that the user ignored (by making a selection first). Similarly, suppose that the user ignores the warning light that says, “Correct change only,” and puts in extra money. Most machines are user-hostile; they will happily swallow the excess money.



**Figure :**Employee Class with Attributes



**Figure :** Employee Objects Tom and Kaitlyn

## Behavior

Behavior is how an object acts and reacts, in terms of its state changeable state of object affect its behavior. In vending machine, if we don't deposit change sufficient for our selection, then the machine will probably do nothing. So behavior of an object is a function of its state as well as the operation performed upon it. The state of an object represents the cumulative results of its behavior.

An operation is some action that one object performs on another in order to elicit a reaction. For example, a client might invoke the operations append and pop to grow and shrink a queue object, respectively. A client might also invoke the operation length, which returns a value denoting the size of the queue object but does not alter the state of the queue itself.

Message passing is one part of the equation that defines the behavior of an object; our definition for behavior also notes that the state of an object affects its behavior as well.

## Operations

An operation denotes a service that a class offers to its clients. A client performs 5 kinds of operations upon an object.

In practice, we have found that a client typically performs five kinds of operations on an object. The three most common kinds of operations are the following:

- **Modifier:** An operation that alters the state of an object.
- **Selector:** An operation that accesses the state of an object but does not alter the state.
- **Iterator:** An operation that permits all parts of an object to be accessed in some well defined order. In queue example operations, clear, append, pop, remove) are modifies, const functions (length, is empty, front location) are selectors.

Two other kinds of operations are common; they represent the infrastructure necessary to create and destroy instances of a class.

- **Constructor:** An operation that creates an object and/or initializes its state.
- **Destructor:** An operation that frees the state of an object and/or destroys the object itself.

## Roles and Responsibilities

A role is a mask that an object wears and so defines a contract between an abstraction and its clients. Responsibilities are meant to convey a sense of the purpose of an object and its place in the system. The responsibilities of an object are all the services it provides for all of the contracts it supports”.

The state and behavior of an object collectively define the roles that an object may play in the world, which in turn fulfill the abstraction’s responsibilities.

### Examples:

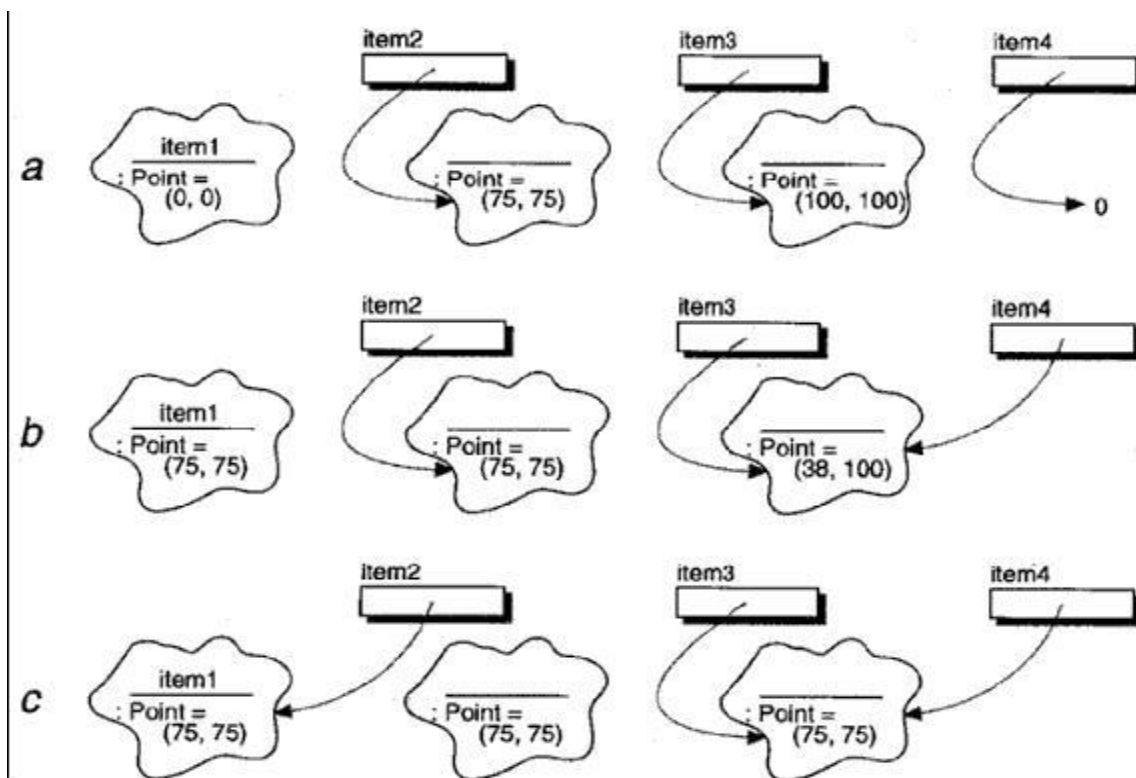
1. A bank account may have the role of a monetary asset to which the account owner may deposit or withdraw money. However, to a taxing authority, the account may play the role of an entity whose dividends must be reported on annually.
2. To a trader, a share of stock represents an entity with value that may be bought or sold; to a lawyer, the same share denotes a legal instrument to which are attached certain rights.
3. In the course of one day, the same person may play the role of mother, doctor, gardener, and movie critic.

### **Identity**

Identity is that property of an object which distinguishes it from all others.

#### **Example:**

Consider a class that denotes a display item. A display item is a common abstraction in all GUI-centric systems: It represents the base class of all objects that have a visual representation on some window and so captures the structure and behavior common to all such objects. Clients expect to be able to draw, select, and move display items, as well as query their selection state and location. Each display item has a location designated by the coordinates x and y.



**Figure :Object Identity**

First declaration creates four names and 3 distinct objects in 4 diff location. Item 1 is the name of a distinct display item object and other 3 names denote a pointer to a display item objects. Item 4 is no such objects, we properly say that item 2 points to a distinct display item object, whose name we may properly refer to indirectly as \* item2. The unique identity (but not necessarily the name) of each object is preserved over the lifetime of the object, even when its state is changed. Copying, Assignment, and Equality Structural sharing takes place when the identity of an object is aliased to a second name.

## → Relationships among Objects

Objects contribute to the behavior of a system by collaborating with one another.

E.g. object structure of an airplane.

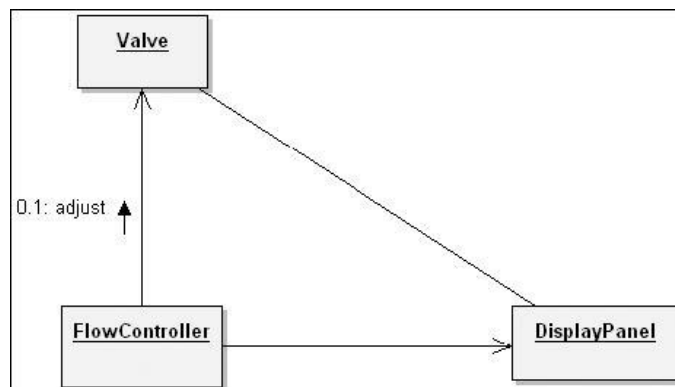
The relationship between any two objects encompasses the assumptions that each makes about the other including what operations can be performed. There are Two kinds of objects relationships are links and aggregation.

### 1. Links

- A link denotes the specific association through which one object (the client) applies the services of another object (the supplier) or through which an object may navigate to another.
- A line between two object icons represents the existence of a path along this path.
- Messages are shown as directed lines representing the direction of message passing between two objects; typically unidirectional, may be bidirectional data flow in either direction across a link.

As a participant in a link, an object may play one of three roles:

- **Controller:** This object can operate on other objects but is not operated on by other objects. In some contexts, the terms active object and controller are interchangeable.
- **Server:** This object doesn't operate on other objects; it is only operated on by other objects.
- **Proxy:** This object can both operate on other objects and be operated on by other objects. A proxy is usually created to represent a real-world object in the domain of the application.



**Figure :Links**

In the above figure, FlowController acts as a controller object, DisplayPanel acts as a server object, and Valve acts as a proxy.

## Visibility

Consider two objects, A and B, with a link between the two. In order for A to send a message to object B, B must be visible to A. Four ways of visibility

- The supplier object is global to the client
- The supplier object is a programmer to some operation of the client
- The supplier object is a part of the client object.
- The supplier object is locally declared object in some operation of the client.

## Synchronization

Wherever one object passes a message to another across a link, the two objects are said to be synchronized. Active objects embody their own thread of control, so we expect their semantics to be guaranteed in the presence of other active objects. When one active object has a link to a passive one, we must choose one of three approaches to synchronization.

1. **Sequential:** The semantics of the passive object are guaranteed only in the presence of a single active object at a time.
2. **Guarded:** The semantics of the passive object are guaranteed in the presence of multiple threads of control, but the active clients must collaborate to achieve mutual exclusion.
3. **Concurrent:** The semantics of the passive object are guaranteed in the presence of multiple threads of control, and the supplier guarantees mutual exclusion.

## 2. Aggregation

Whereas links denote peer to peer or client/supplier relationships, aggregation denotes a whole/part hierarchy, with the ability to navigate from the whole (also called the aggregate) to its parts. Aggregation is a specialized kind of association. Aggregation may or may not denote physical containment. E.g. airplane is composed of wings, landing gear, and so on. This is a case of physical containment. The relationship between a shareholder and her shares is an aggregation relationship that doesn't require physical containment.

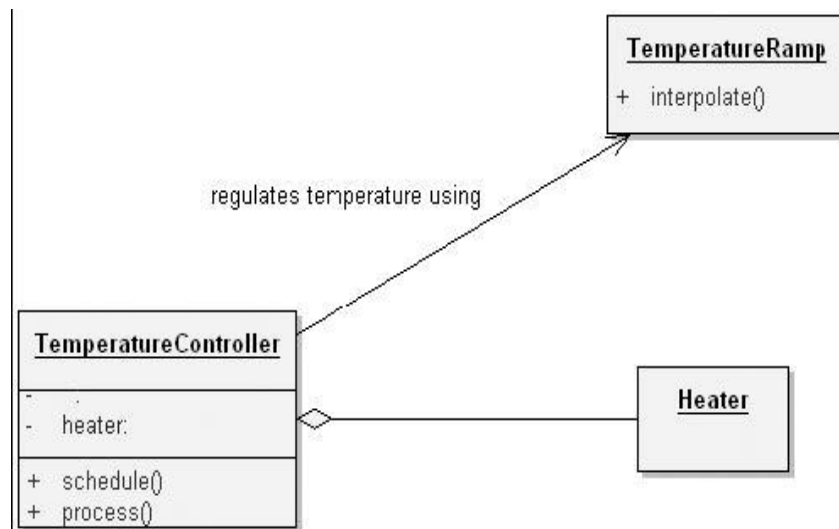


Figure : Aggregation



## → The Nature of the class

A class is a set of objects that share a common structure, common behavior and common semantics. A single object is simply an instance of a class. Object is a concrete entity that exists in time and space but class represents only an abstraction. A class may be an object is not a class.

### **Interface and Implementation:**

The interface of a class provides its outside view and therefore emphasizes the abstraction while hiding its structure and secrets of its behavior. The interface primarily consists of the declarations of all the operators applicable to instance of this class, but it may also include the declaration of other classes, constants variables and exceptions as needed to complete the abstraction. The implementation of a class is its inside view, which encompasses the secrets of its behavior. The implementation of a class consists of the class. Interface of the class is divided into following four parts.

- **Public:** a declaration that is accessible to all clients
- **Protected:** a declaration that is accessible only to the class itself and its subclasses
- **Private:** a declaration that is accessible only to the class itself
- **Package:** a declaration that is accessible only by classes in the same package.

## → Relationship among Classes

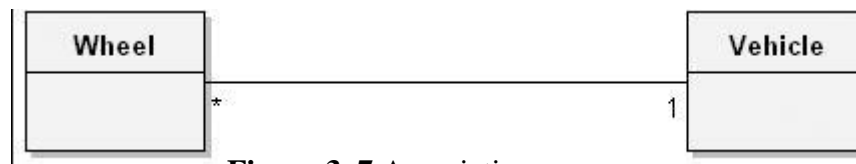
We establish relationships between two classes for one of two reasons. First, a class relationship might indicate some kind of sharing. Second, a class relationship might indicate some kind of semantic connection.

There are four basic kinds of class relationships. They are

1. Association
2. Aggregation
- 3 Inheritance (Generalization)
4. Dependency

### **1. Association:**

An association is a structural relationship that describes a set of links, a link being connection among objects. Of the different kinds of class relationships, associations are the most general. The identification of associations among classes is describing how many classes/objects are taking part in the relationship. As an example for a vehicle, two of our key abstractions include the vehicle and wheels. As shown in Figure , we may show a simple association between these two classes: the class Wheel and the class Vehicle.



**Figure 3–7 Association**

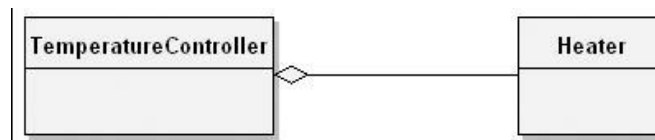
## Multiplicity/Cardinality

This multiplicity denotes the cardinality of the association. There are three common kinds of multiplicity across an association:

1. One-to-one
2. One-to-many
3. Many-to-many

## 2. Aggregation

Aggregation is a special kind of association, representing a structural relationship between a whole and its parts. Aggregation relationships among classes have a direct parallel to aggregation relationships among the objects corresponding to these classes.

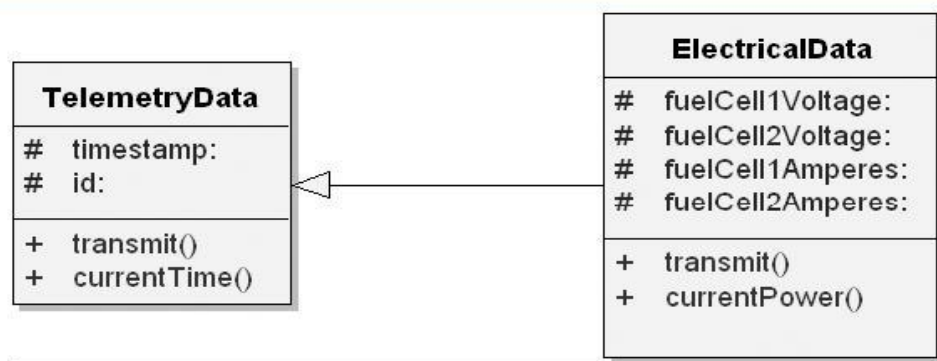


**Figure :** Aggregation

As shown in Figure, the class Temperature Controller denotes the whole, and the class Heater is one of its parts.

## 3. Inheritance

Inheritance, perhaps the most semantically interesting of the concrete relationships, exists to express generalization/specialization relationships. Inheritance is a relationship among classes wherein one class shares the structure and/or behavior defined in one (single inheritance) or more (multiple inheritance) other classes. Inheritance means that subclasses inherit the structure of their superclass.

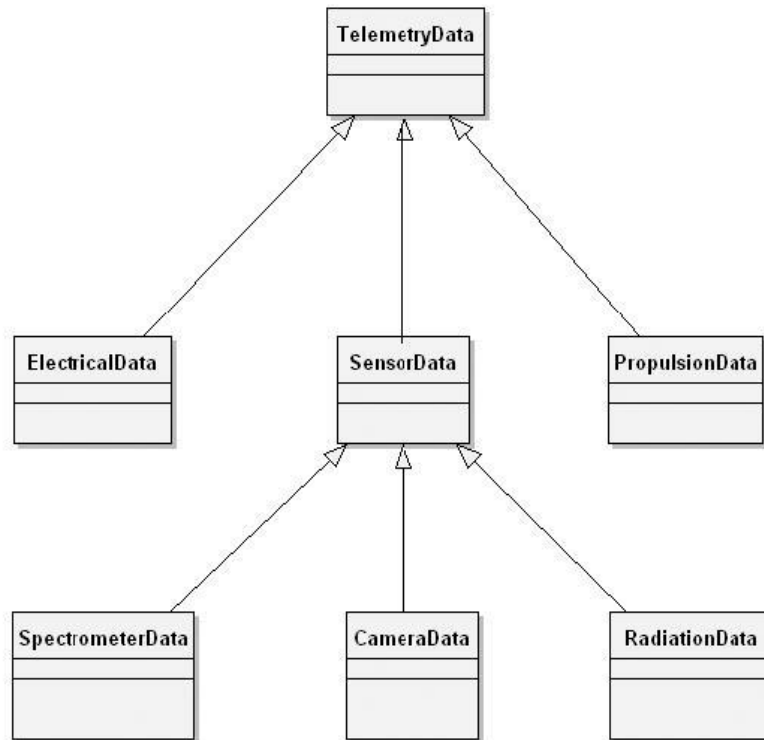


**Figure :** ElectricalData Inherits from the Superclass TelemetryData

As for the class ElectricalData, this class inherits the structure and behavior of the class TelemetryData but adds to its structure (the additional voltage data), redefines its behavior (the function transmit) to transmit the additional data, and can even add to its behavior (the function currentPower, a function to provide the current power level).

**Single Inheritance:**

The derived class having only one base class is called single inheritance. It is a relationship among classes where in one class shares the structure and/or behavior defined one class.

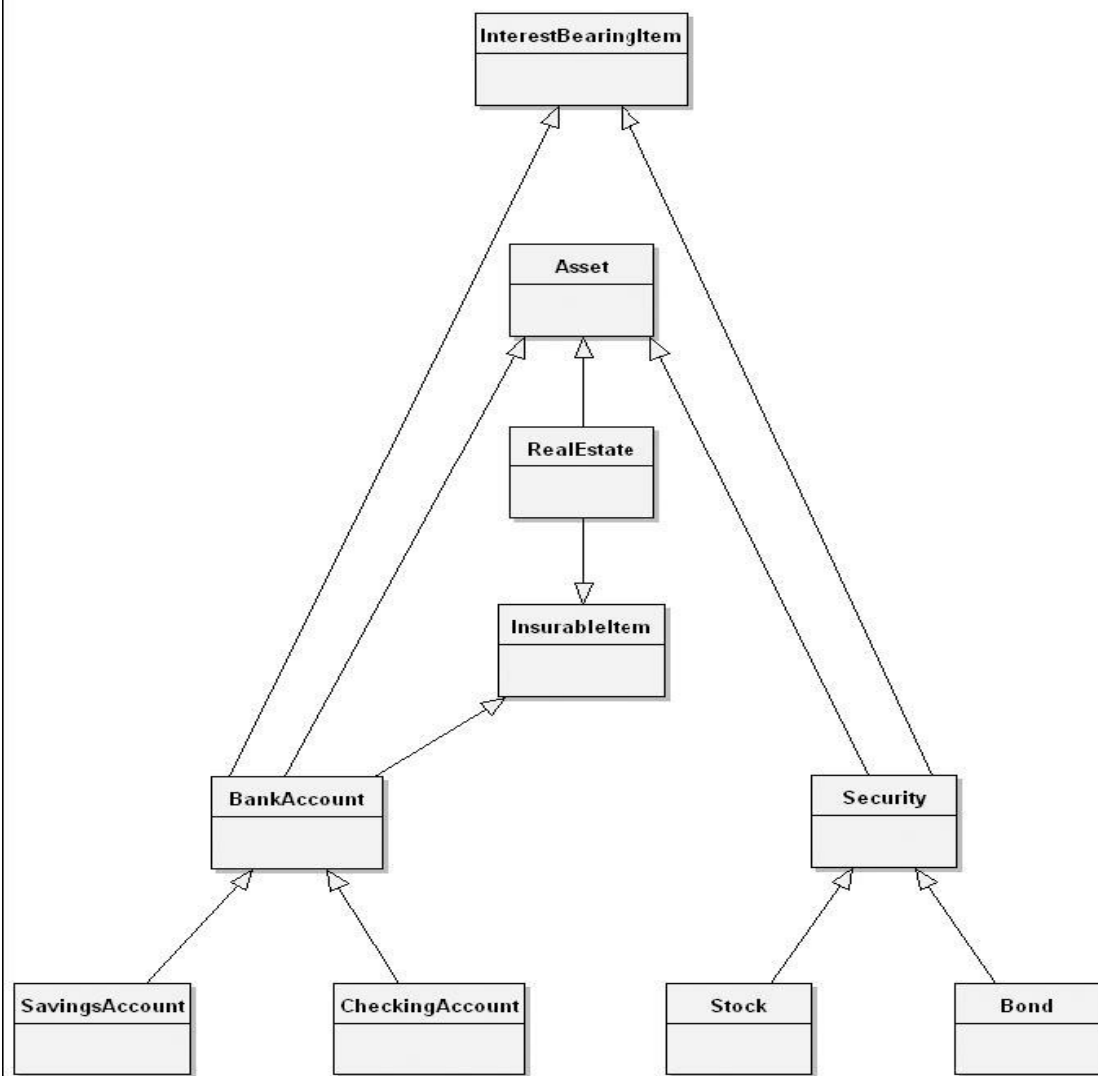


**Fig:** Single Inheritance

Figure illustrates the single inheritance relationships deriving from the superclass TelemetryData. Each directed line denotes an “is a” relationship. For example, CameraData “is a” kind of SensorData, which in turn “is a” kind of TelemetryData.

**Multiple Inheritance:**

The derived class having more than one base class is known as multiple inheritance. It is a relationship among classes where in one class shares the structure and/or behavior defined more classes.

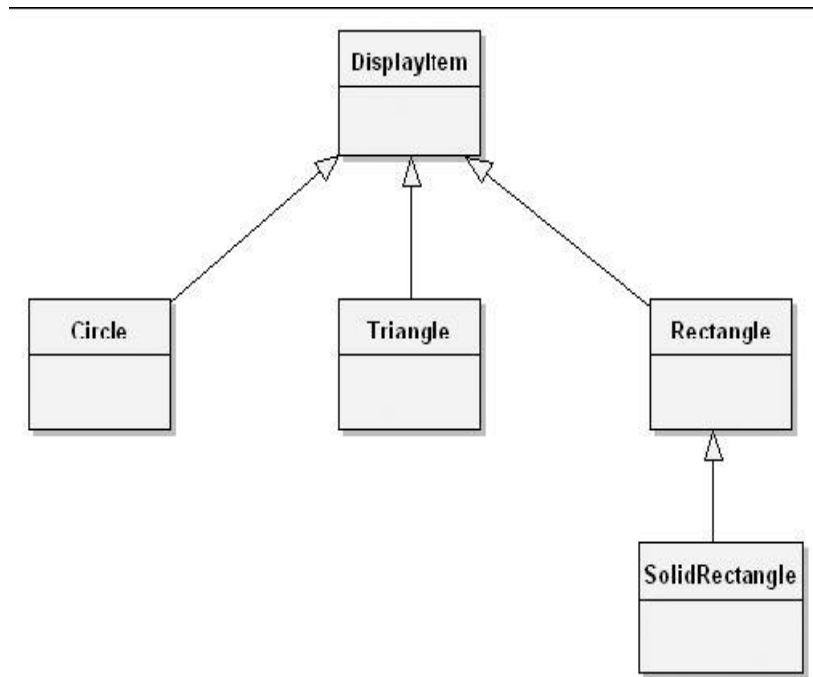


**Figure** Multiple Inheritance

Unfortunately, single inheritance is not expressive enough to capture this lattice of relationships, so we must turn to multiple inheritance. Above Figure illustrates such a class structure. Here we see that the class `Security` is a kind of `Asset` as well as a kind of `InterestBearingItem`. Similarly, the class `BankAccount` is a kind of `Asset`, as well as a kind of `InsurableItem` and `InterestBearingItem`.

## Polymorphism

Polymorphism is an ability to take more than one form. It is a concept in type theory wherein a name may denote instances of many different classes as long as they are related by some common superclass. Any object denoted by this name is thus able to respond to some common set of operations in different ways. With polymorphism, an operation can be implemented differently by the classes in the hierarchy.



**Figure : Polymorphisms**

Consider the class hierarchy in Figure, which shows the base class **DisplayItem** along with three subclasses named **Circle**, **Triangle**, and **Rectangle**. **Rectangle** also has one subclass, named **SolidRectangle**. In the class **DisplayItem**, suppose that we define the instance variable **theCenter** (denoting the coordinates for the center of the displayed item), along with the following operations:

- **draw**: Draw the item.
- **move**: Move the item.
- **location**: Return the location of the item.

The operation **location** is common to all subclasses and therefore need not be redefined, but we expect the operations **draw** and **move** to be redefined since only the subclasses know how to draw and move themselves.

### → **The interplay of classes and relationships:**

- Classes and objects are separate yet intimately related concepts.
- Specifically, every object is the instance of some class, and every class has zero or more instances.
- For all applications, classes are static; therefore, their existence, semantics, and relationships are fixed prior to the execution of a program.
- Similarly, the class of most objects is static, meaning that once an object is created, its class is fixed.
- Objects are typically created and destroyed at a furious rate during the lifetime of an application

### **Relationship between classes and objects:**

For example, consider the classes and objects in the implementation of an air traffic control system. Some of the more important abstractions include planes, flight plans, runways and air spaces. These classes and objects are relatively static. Conversely,

the instances of these classes are dynamic. At a fairly slow rate, new run ways are built and old ones are deactivated.

## **The Role of Classes and Objects in Analysis and Design**

During analysis and the early stages of design, the developer has two primary tasks

1. Identify the classes that form the vocabulary of the problem domain
2. Invent the structures whereby sets of objects work together to provide the behaviors that satisfy the requirements of the problem

Collectively, we call such classes and objects the *key abstractions* of the problem, and We call these cooperative structures the *mechanisms* of the implementation.

## **Part-II(Classification)**

### **—→ Importance of proper classification**

Classification is the means whereby we order knowledge. There is no any golden path to classification. Classification and object oriented development: The identification of classes and objects is the hardest part of object oriented analysis and design, identification involves both discovery and invention. Discovery helps to recognize the key abstractions and mechanisms that form the vocabulary of our problem domain. Through invention, we desire generalized abstractions as well as new mechanisms that specify how objects collaborate discovery and inventions are both problems of classifications.

Intelligent classification is actually a part of all good science class of should be meaningful is relevant to every aspect of object oriented design classify helps us to identify generalization, specialization, and aggregation hierarchies among classes classify also guides us making decisions about modularizations.

### **The difficulty of classification**

Examples of classify: Consider start and end of knee in leg in recognizing human speech, how do we know that certain sounds connect to form a word, and aren't instead a part of any surrounding words?

In word processing system, is character class or words are class? Intelligent classification is difficult e.g. problems in classify of biology and chemistry until 18<sup>th</sup> century, organisms were arranged from the most simple to the most complex, human at top of the list. In mid 1970, organisms were classified according to genus and species. After a century later, Darwin's theory came which was depended upon an intelligent classification of species. Category in biological

taxonomy is the kingdom, increased in order from phylum, subphylum class, order, family, genus and finally species. Recently classify has been approached by grouping organisms that share a common generic heritage i.e. classify by DNA. DNA in useful in distinguishing organisms that are structurally similar but genetically very different classify depends on what you want classification to do. In ancient times, all substances were through to be sure ambulation of earth, fire, air and water. In mid 1960s – elements were primitive abstractive of chemistry in 1869 periodic law came.

### **The incremental and iterative nature of classification**

Intelligent classification is intellectually hard work, and that it best comes about through an incremental and iterative process. Such processes are used in the development of software technologies such as GUI, database standards and programming languages. The useful

solutions are understood more systematically and they are codified and analyzed. The incremental and iterative nature of classification directly impacts the construction of class and object hierarchies in the design of a complex software system. In practice, it is common to assert in the class structure early in a design and then revise this structure over time. Only at later in the design, once clients have been built that use this structure, we can meaningfully evaluate the quality of our classification. On the basis of this experience, we may decide to create new subclasses from existing ones (derivation). We may split a large class into several smaller ones (factorization) or create one large class by uniting smaller ones (composition). Classify is hard because there is no such as a perfect classification (classify are better than others) and intelligent classify requires a tremendous amount of creative insight.

## —→ **Identifying classes and objects**

Classical and modern approaches: There are three general approaches to classifications.

- Classical categorization
- Conceptual clustering
- Prototypal theory

### **1. Classical categorizations**

All the entities that have a given property or collection of properties in common forms a category. Such properties are necessary and sufficient to define the category. i.e. married people constitute a category i.e. either married or not. The values of this property are sufficient to decide to which group a particular person belongs to the category of tall/short people, where we can agree to some absolute criteria. This classification came from Plato and then from Aristotle's classification of plants and animals.

This approach of classification is also reflected in modern theories of child development. Around the age of one, child typically develops the concept of object permanence, shortly thereafter, the child acquires skill in classifying these objects, first using basic category such as dogs, cats and toys.

### **2. Conceptual clustering**

It is a more modern variation of the classical approach and largely derives from attempts to explain how knowledge is represented in this approach, classes are generated by first formulating conceptual description of these classes and then classifying the entities according to the descriptions. e.g. we may state a concept such as "a love song". This is a concept more than a property, for the "love songness" of any song is not something that may be measured empirically. However, if we decide that a certain song is more of a love song than not, we place it in this category. thus this classify represents more of a probabilistic clustering of objects and objects may belong to one or more groups, in varying degree of fitness conceptual clustering makes absolute judgments of classify by focusing upon the best fit.

### **3. Prototype theory**

It is more recent approach of classify where a class of objects is represented by a prototypical object, an object is considered to be a member of this class if and only if it resembles this prototype in significant ways. e.g. category like games, not in classical since no single common properties shared by all games, e.g. classifying chairs (beanbag chairs, barber chairs, in prototypes theory, we group things according to the degree of their relationship to concrete prototypes.

These approaches to classify provide the theoretical foundation of objected analysis by which we identify classes and objects in order to design a complex software system.

## **Object oriented Analysis**

The boundaries between analysis and design are fuzzy, although the focus of each is quite distinct. In analysis, we seek to model the world by discovering. The classes and objects that form the vocabulary of the problem domain and in design, we invent the abstractions and mechanisms that provide the behavior that this model requires. Following are some approaches for analysis that are relevant to object oriented system.

### **Classical approaches**

It is one of approaches for analysis which derive primarily from the principles of classical categorization. e.g. Shlaer and Mellor suggest that classes and objects may come from the following sources:

- Tangible things, cars, pressure sensors
- Roles – Mother, teacher, politician
- Events – landing, interrupt
- Interactions – meeting

From the perspective of database modeling, Ross offers the following list:

- (i) People – human who carry out some function
- (ii) Places – Areas set for people or thing
- (iii) Things – Physical objects (tangible)
- (iv) Organizations – organized collection of people resources
- (v) Concepts – ideas
- (vi) Events – things that happen

Coad and Yourdon suggest another set of sources of potential objects.

- (i) Structure
- (ii) Dences
- (iii) Events remembered (historical)
- (iv) Roles played (of users)
- (v) Locations (office, sites)
- (vi) Organizational units (groups)

### **Behavior Analysis**

Dynamic behavior also be one of the primary source of analysis of classes and objects things can are grouped that have common responsibilities and form hierarchies of classes (including superclasses and subclasses). System behaviors of system are observed. These behaviors are assigned to parts of system and tried to understand who initiates and who participates in these behaviors. A function point is defined as one and user business functions and represents some kind of output, inquiry, input file or interface.

### **Domain Analysis**

Domain analysis seeks to identify the classes and objects that are common to all applications within a given domain, such as patient record tracking, compilers, missile systems etc. Domain analysis defined as an attempt to identify the objects, operations and, relationships that are important to particular domain.

More and Bailin suggest the following steps in domain analysis.

- i) Construct a strawman generic model of the domain by consulting with domain expert.
- ii) Examine existing system within the domain and represent this understanding in



a common format.

- iii) Identify similarities and differences between the system by consulting with domain expert.
- iv) Refine the generic model to accommodate existing systems.

**Vertical domain Analysis:** Applied across similar applications.

**Horizontal domain Analysis:** Applied to related parts of the same application domain expert is like doctor in a hospital concerned with conceptual classification.

### **Use case Analysis**

Earlier approaches require experience on part of the analyst such a process is neither deterministic nor predictably successful. Use case analysis can be coupled with all three of these approaches to derive the process of analysis in a meaningful way. Use case is defined as a particular form pattern or exemplar some transaction or sequence of interrelated events. Use case analysis is applied as early as requirements analysis, at which time end users, other domain experts and the development team enumerate the scenarios that are fundamental to system's operation. These scenarios collectively describe the system functions of the application analysis then proceeds by a study of each scenario. As the team walks through each scenario, they must identify the objects that participate in the scenario, responsibilities of each object and how those objects collaborate with other objects in terms of the operations each invokes upon the other.

### **CRC cards**

CRC are a useful development tool that facilitates brainstorming and enhances communication among developers. It is 3 x 5 index card (class/Responsibilities/collaborators i.e. CRC) upon which the analyst writes in pencil with the name of class (at the top of card), its responsibilities

(on one half of the card) and its collaborators (on the other half of the card). One card is created for each class identified as relevant to the scenario. CRC cards are arranged to represent generalization/specialization or aggregation hierarchies among the classes.

### **Informal English Description**

Proposed by Abbott. It is writing an English description of the problem (or a part of a problem) and then underlining the nouns and verbs. Nouns represent candidate objects and the verbs represent candidate operations upon them. it is simple and forces the developer to work in the vocabulary of the problem space.

### **Structured Analysis**

Same as English description as an alternative to the system, many CASE tools assists in modeling of the system. In this approach, we start with an essential model of the system, as described by data flow diagrams and other products of structured analysis. From this model we may proceed to identify the meaningful classes and objects in our problem domain in 3 ways.

- Analyzing the context diagrams, with list of input/output data elements; think about what they tell you or what they describe e.g. these make up list of candidate objects.
- Analyzing data flow domains, candidate objects may be derived from external entities, data stores, control stores, control transformation, candidate classes derive from data flows and candidate flows.
- By abstraction analysis: In structured analysis, input and output data are examined and followed inwards until they reach the highest level of abstraction.

## → **Key abstractions and mechanisms**

A key abstraction is a class or object that forms part of the vocabulary of the problem domain. The primary value of identifying such abstractions is that they give boundaries to our problems. They highlight the things that are in the system and therefore relevant to our design and suppress the things that are outside of system.

### **Identification of Abstractions:**

identification of key abstraction involves two processes. Discovery and invention through discovery we come to recognize the abstraction used by domain experts. If through inventions, we create new classes and objects that are not necessarily part of the problem domain. A developer of such a system uses these same abstractions, but must also introduce new ones such as databases, screen managers, lists queues and so on. These key abstractions are artifacts of the particular design, not of the problem domain.

### **Refining key abstractions**

Once we identify a certain key abstraction as a candidate, we must evaluate it. Programmer must focus on questions. How we objects of this class created? What operations can be done on such objects? If there are not good answers to such questions, then the problem is to be thought again and proposed solution is to be found out instead of immediately starting to code among the problems placing classes and objects at right levels of abstraction is difficult. Sometimes we may find a general subclass and so may choose to move it up in the class structure, thus increasing the degree of sharing. This is called class promotion. Similarly, we may find a class to be too general, thus making inheritance by a subclass difficult because of the large semantic gap. This is called a grain size conflict.

Naming conventions are as follows:

- Objects should be named with proper noun phrases such as the sensor or simply shapes.
- Classes should be named with common noun phrases, such as sensor or shapes.
- Modifier operations should be named with active verb phrases such as draw, moveleft.
- Selector operations should imply a query or be named with verbs of the form "to be" e.g. is open, extent of.

### **Identifying Mechanisms**

A mechanism is a design decision about how collection of objects cooperates. Mechanisms represent patterns of behavior e.g. consider a system requirement for an automobile: pushing the accelerator should cause the engine to run faster and releasing the accelerator should cause the engine to run slower. Any mechanism may be employed as long as it delivers the required behavior and thus which mechanism is selected is largely a matter of design choice. Any of the following design might be considered.

- A mechanical linkage from the acceleration to the (the most common mechanism)
- An electronic linkage from a pressure sensor below the accelerator to a computer that controls the carburetor (a drive by wire mechanism)
- No linkage exists; the gas tank is placed on the roof of the car and gravity causes fuel to flow to the engine. Its rate of flow is regulated by a clip around the fuel the pushing on the accelerator pedal eases tension on the clip, causing the fuel to flow faster (a low cost mechanism)

### **Examples of mechanisms:**

Consider the drawing mechanism commonly used in graphical user interfaces. Several objects must collaborate to present an image to a user: a window, a new, the model being

viewed and some client that knows when to display this model. The client first tells the window to draw itself. Since it may encompass several subviews, the window next tells each if its subviews to draw them. Each subview in turn tells the model to draw itself ultimately resulting in an image shown to the user.

### **PREVIOUS QUESTIONS**

1. Discuss about possible relationships formed among the objects with example.
2. Describe how to identify classes and objects in detail.
3. Explain interplay of classes and objects.
4. State importance of proper classification using an example.
5. Describe Key abstraction.

### **Two Marks Questions**

1. What are the different class associations?
2. State about relationships among objects.
3. What are the advantages of classification?
4. Explain CRC card. Write uses of crc cards.
5. Write about association.
6. What is the main impact of object oriented approach?
7. Enumerate three most common operations on object.
8. Examine the nature of class.

## UNIT-III

**Introduction to UML:** Why we model, Conceptual model of UML, Architecture, Classes, Relationships, Common mechanisms, Class diagrams, object diagrams.

---

### → **Why we model**

#### **The importance of Modeling:**

A model is a simplification of reality. A model provides the blueprints of a system. Every system may be described from different aspects using different models, and each model is therefore a semantically closed abstraction of the system.

A model may be structural, emphasizing the organization of the system, or it may be behavioral, emphasizing the dynamics of the system. We build models so that we can better understand the system we are developing.

Through modeling, we achieve four aims.

1. Models help us to visualize a system as it is or as we want it to be.
2. Models permit us to specify the structure or behavior of a system.
3. Models give us a template that guides us in constructing a system.
4. Models document the decisions we have made.

Modeling is not for big systems. Even the software equivalent of a dog house can benefit from some modeling. For example if you want to build a dog house, you can start with a pile of lumber, some nails, and a basic tools, such as a hammer, saw, and a tape measure. In few hours, with little prior planning, you will likely end up with a dog house that's reasonably functional. Finally you will be happy and get a less demanding dog.

If you want to build a house for your family, you can start with a pile of lumber, some nails, and a basic tools. But it's going to take you a lot longer, and your family will certainly be more demanding than the dog. If you want to build a quality house that meets the needs of your family and you will need to draw some blueprints.

#### **Principles of Modeling:**

UML is basically a modeling language; hence its principles will also be related to modeling concepts. Here are few basic principles of UML.

**First:** "The choice of what models to create has a profound influence on how a problem is attacked and how a solution is shaped"

In other words ,choose your models well. The right models will brilliantly illuminate the most wicked development problems. The wrong models will mislead you, causing you to focus on irrelevant issues.

**Second:** " Every model may be expressed at different levels of precision ".

Best approach to a given problem results in a best model. If the problem is complex mechanized level of approach & if the problem is simple decent approach is followed.

**Third:** "The best models are connected to reality."

The model built should have strong resemblance with the system.

**Fourth:** " No single model is sufficient. Every nontrivial system is best approached through a small set of nearly independent models."

If you constructing a building, there is no single set of blueprints that reveal all its details. At the very least, you will need floor plans,elevations,electrical plans, heating plans, and plumbing plans.

## Object-Oriented Modeling:

In software , there are several ways to approaches a model. The two most common ways are

1. Algorithmic perspective
2. Object-Oriented perspective

### 1. Algorithmic perspective:

In this approach, the main building blocks of all software is the procedure or function.This view leads developers to focus on issues of control and decomposition of larger algorithms into smaller ones.

### 2. Object-Oriented perspective:

In this approach, the main building blocks of all software is the object or class. Simply put, an object is a thing. A class is a description of a set of common objects. Every object has identity, state and behavior.

For example, consider a simple a three-tier -architecture for a billing system, involving a user interface ,middleware, and a data base. In the user interface, you will find concrete objects, such as buttons, menus, and dialog boxes. In the database, you will find concrete objects ,such as tables. In the middle layer ,you will find objects such as transitions and business rules.

## →A Conceptual Model of the UML

To understand the UML, we need to form a conceptual model of the language, and this requires learning three major elements:

- \* Basic Building blocks of the UML
- \* Rules of the UML
- \* Common mechanisms in the UML.

## **\*\*Basic Building Blocks of the UML:**

The vocabulary of the UML encompasses three kinds of building blocks:

1. Things in the UML
2. Relationships in the UML
3. Diagrams in the UML

### **1.Things in the UML: -**

Things are the abstractions that are first-class citizens in a model. There are four kinds of things in the UML:

1. Structural things
2. Behavioral things
3. Grouping things
4. Annotational things

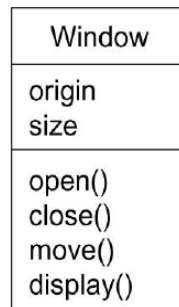
#### **1. Structural things:**

Structural things are the nouns of UML models. These are the mostly static parts of a model. There are seven kinds of structural things. They are

- a. Class
- b. Interface
- c. Collaboration
- d. Use case
- e. Activity Class
- f. Component
- g. Node

##### **a) Class:**

A **class** is a description of a set of objects that share the same attributes, operations, relationships, and semantics. A class implements one or more interfaces. Graphically, a class is rendered as a rectangle, usually including its name, attributes, and operations.



##### **b) Interface:**

An interface is a collection of operations that specify a service of a class or component. Graphically, an interface is rendered as a circle together with its name.



##### **c) Collaboration:**

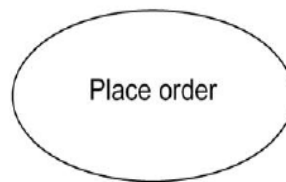
Collaboration defines an interaction and is a society of roles and other elements that work together to provide some cooperative behavior that's bigger than the sum of all the

elements. Graphically, a collaboration is rendered as an ellipse with dashed lines, usually including only its name.



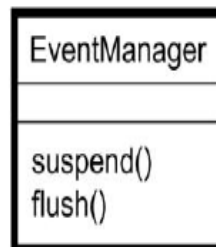
**d) Use case:**

A use case is a description of set of sequence of actions that a system performs that yields an observable result of value to a particular actor. A use case is used to structure the behavioral things in a model. A use case is realized by a collaboration. Graphically, a use case is rendered as an ellipse with solid lines, usually including only its name.



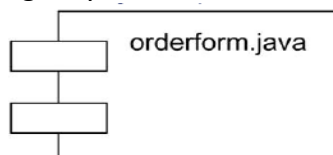
**e)Active class**

An active class is a class whose objects own one or more processes or threads and therefore can initiate control activity. Graphically, an active class is rendered just like a class, but with heavy lines, usually including its name, attributes, and operations.



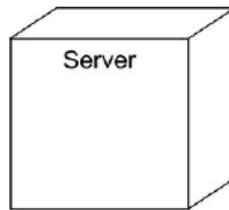
**f) Component:**

A Component is a physical and replaceable part of a system that conforms to and provides the realization of a set of interfaces. Graphically, a component is rendered as a rectangle with tabs, usually including only its name.



**g) Node:**

A **node** is a physical element that exists at run time and represents a computational resource, generally having at least some memory and, often, processing capability. Graphically, a node is rendered as a cube, usually including only its name.



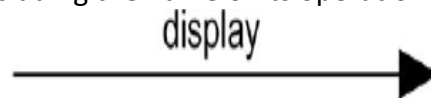
## 2. Behavioral things:

Behavioral things are the dynamic parts of UML models. These are the verbs of a model, representing behavior over time and space. There are two primary kinds of behavioral things. They are

- a) Interaction
- b) State machine

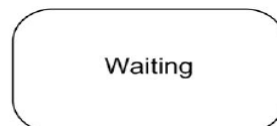
### a) Interaction

Interaction is a behavior that comprises a set of messages exchanged among a set of objects within a particular context to accomplish specific purpose. An interaction involves a number of other elements, including messages, action sequences (the behavior invoked by a message), and links (the connection between objects). Graphically, a message is rendered as a **directed line**, almost always including the name of its operation.



### b) State machine

A State machine a number of other elements, including states, transitions (the flow from state to state), events (things that trigger a transition), and activities (the response to a transition). Graphically, a state is rendered as a **rounded rectangle**, usually including its name and its sub states.

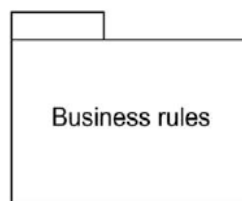


## 3. Grouping things:

Grouping things are the organizational parts of UML models. These are the boxes into which a model can be decomposed. There is one primary kind of grouping thing, namely, packages.

### package:

A package is a general-purpose mechanism for organizing elements into groups. Structural things, behavioral things, and even other grouping things may be placed in a package. Graphically, a package is rendered as a **tabbed folder**, usually including only its name and, sometimes, its contents.



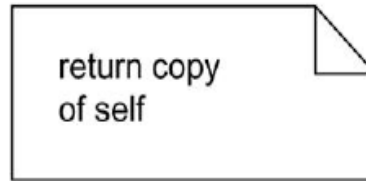
## 4. Annotational things:



Annotational things are the explanatory parts of UML models. These are the comments you may apply to describe, illuminate, and remark about any element in a model. There is one primary kind of annotational thing, called a note.

**Note:**

A note is simply a symbol for rendering constraints and comments attached to an element or a collection of elements. Graphically, a note is rendered as a rectangle with a **dog-eared corner**, together with a textual or graphical comment.



## **1. Relationships in the UML:**

Relationships are used to connect things. There are four kinds of relationships in the UML:

1. Dependency
2. Association
3. Generalization
4. Realization

### **1. Dependency:**

A dependency is a semantic relationship between two things in which a change to one thing (the independent thing) may affect the semantics of the other thing (the dependent thing). Graphically, a dependency is rendered as a dashed line, possibly directed, and occasionally including a label.



### **2. Association:**

An **association** is a structural relationship that describes a set of links, a link being a connection among objects. Aggregation is a special kind of association, representing a structural relationship between a whole and its parts. Graphically, an association is rendered as a solid line, possibly directed, occasionally including a label, and often containing other adornments, such as multiplicity and role names.



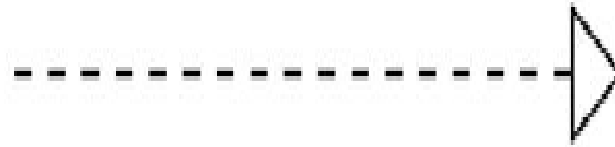
### **3. Generalization:**

A Generalization is a specialization/generalization relationship in which objects of the specialized element (the child) are substitutable for objects of the generalized element (the parent). In this way, the child shares the structure and the behavior of the parent. Graphically, a generalization relationship is rendered as a solid line with a hollow arrowhead pointing to the parent.



#### 4. Realization:

A **Realization** is a semantic relationship between classifiers, wherein one classifier specifies a contract that another classifier guarantees to carry out. Graphically, a realization relationship is rendered as a cross between a generalization and a dependency relationship.



### 3. Diagrams in the UML:-

A diagram is the graphical presentation of a set of elements, most often rendered as a connected graph of vertices (things) and arcs (relationships). The UML includes nine such diagrams:

- |                          |                        |
|--------------------------|------------------------|
| 1. Class diagram         | 2. Object diagram      |
| 3. Use case diagram      | 4. Sequence diagram    |
| 5. Collaboration diagram | 6. State chart diagram |
| 7. Activity diagram      | 8. Component diagram   |
|                          | 9. Deployment diagram  |

#### 1. Class diagram:

A class diagram shows a set of classes, interfaces, and collaborations and their relationships. These diagrams are the most common diagram found in modeling object-oriented systems. Class diagrams address the static design view of a system.

#### 2. Object diagram:

An object diagram shows a set of objects and their relationships. Object diagrams represent static snapshots of instances of the things found in class diagrams. These diagrams address the static design view or static process view of a system.

#### 3. use case diagram:

A use case diagram shows a set of use cases and actors (a special kind of class) and their relationships. Use case diagrams address the static use case view of a system. These diagrams are especially important in organizing and modeling the behaviors of a system.

#### 4. Sequence diagram:

A sequence diagram is an interaction diagram that emphasizes the time-ordering of messages; Interaction diagrams address the dynamic view of a system. Sequence diagrams and collaboration diagrams are isomorphic, meaning that you can take one and transform it into the other.

#### 5. collaboration diagram:

A **collaboration diagram** is an interaction diagram that emphasizes the structural organization of the objects that send and receive messages. Interaction diagrams address the dynamic view of a system.

Sequence diagrams and collaboration diagrams are isomorphic, meaning that you can take one and transform it into the other.

#### 6. statechart diagram:

A statechart diagram shows a state machine, consisting of states, transitions, events, and activities. Statechart diagrams address the dynamic view of a system.

#### 7. Activity diagram

An activity diagram is a special kind of a state chart diagram that shows the flow from activity to activity within a system. Activity diagrams address the dynamic view of a system.

#### 8. component diagram:

A component diagram shows the organizations and dependencies among a set of components. Component diagrams address the static implementation view of a system.

9. deployment diagram: A deployment diagram shows the configuration of run-time processing nodes and the components that live on them. Deployment diagrams address the static deployment view of architecture.

### → Rules of the UML:-

Like any language, the UML has a number of rules that specify what a well-formed model should look like. A well-formed model is one that is semantically self-consistent and in harmony with all its related models. The UML has semantic rules for

- **Names** : What you can call things, relationships, and diagrams
- **Scope** : The context that gives specific meaning to a name
- **Visibility** : How those names can be seen and used by others
- **Integrity** : How things properly and consistently relate to one another
- **Execution** : What it means to run or simulate a dynamic model

Models built during the development of a software-intensive system tend to evolve and may be viewed by many stakeholders in different ways and at different times. For this reason, it is common for the development team to not only build models that are well-formed, but also to build models that are

- Elided Certain elements are hidden to simplify the view
- Incomplete Certain elements may be missing
- Inconsistent The integrity of the model is not guaranteed.

### → Common Mechanisms in the UML:

UML is made simpler by the presence of four common mechanisms that apply consistently throughout the language. They are:

1. Specifications
2. Adornments
3. Common divisions
4. Extensibility mechanisms

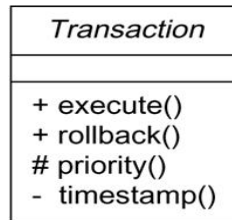
#### 1. Specifications

The UML is more than just a graphical language. Rather, behind every part of its graphical notation there is a specification that provides a textual statement of the syntax and semantics of that building block. For example, behind a class icon is a specification that

provides the full set of attributes, operations (including their full signatures), and behaviors that the class embodies;

## **2.Adornments**

Most elements in the UML have a unique and direct graphical notation that provides a visual representation of the most important aspects of the element. A class's specification may include other details, such as whether it is abstract or the visibility of its attributes and operations. Many of these details can be rendered as graphical or textual adornments to the class's basic rectangular notation.



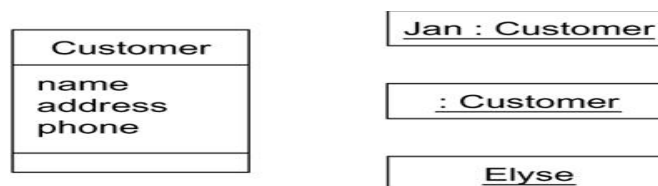
For example, Figure shows a class, adorned to indicate that it is an abstract class with two public, one protected, and one private operation. Every element in the UML's notation starts with a basic symbol, to which can be added a variety of adornments specific to that symbol.

## **3.Common Divisions**

In modeling object-oriented systems, the world often gets divided in at least a couple of ways.

### **1. class and object:**

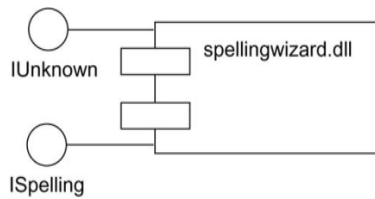
A class is an abstraction; an object is one concrete manifestation of that abstraction. In the UML, you can model classes as well as objects, as shown in Figure



In this figure, there is one class, named *Customer*, together with three objects: *Jan* (which is marked explicitly as being a *Customer* object), *:Customer* (an anonymous *Customer* object), and *Elyse* (which in its specification is marked as being a kind of *Customer* object, although it's not shown explicitly here). Graphically, the UML distinguishes an object by using the same symbol as its class and then simply underlining the object's name.

### **2. interface and implementation.:**

An interface declares a contract, and an implementation represents one concrete realization of that contract, responsible for faithfully carrying out the interface's complete semantics. In the UML, you can model both interfaces and their implementations, as shown in Figure



In this figure, there is one component named spellingwizard.dll that implements two interfaces, IUnknown and ISpelling.

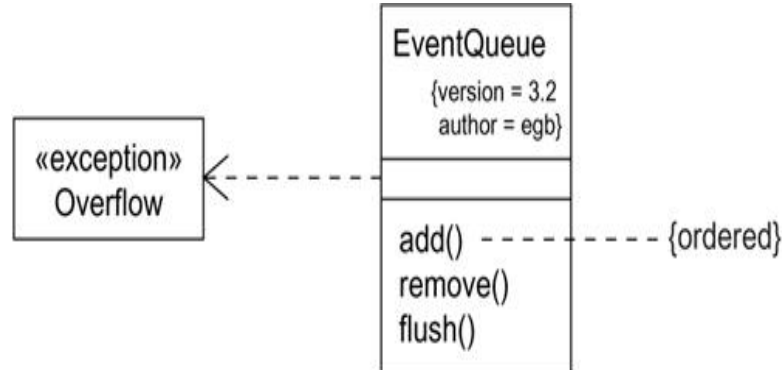
#### **4.Extensibility Mechanisms**

The UML provides a standard language for writing software blueprints, but it is not possible for one closed language to ever be sufficient to express all possible nuances of all models across all domains across all time. The UML's extensibility mechanisms include

1. Stereotypes
2. Tagged values
3. Constraints

##### **1. Stereotypes**

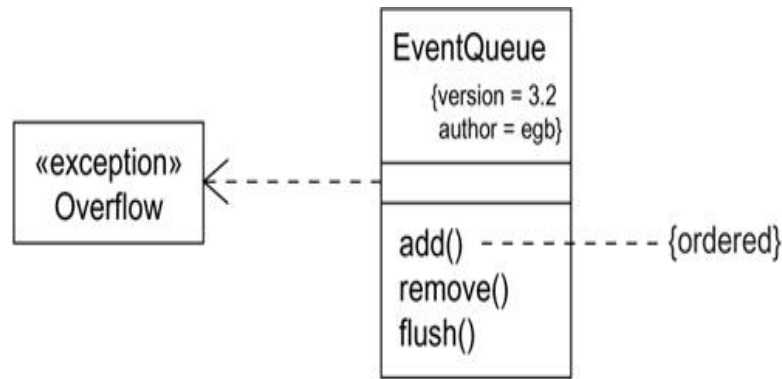
A *stereotype* extends the vocabulary of the UML, allowing you to create new kinds of *building* blocks that are derived from existing ones but that are specific to your problem. For example, if you are working in a programming language, such as Java or C++, you will often want to model exceptions. In these languages, exceptions are just classes. You can make exceptions first class citizens in your models, meaning that they are treated like basic building blocks, by marking them with an appropriate stereotype, as for the class Overflow in Figure.



##### **2. Tagged values**

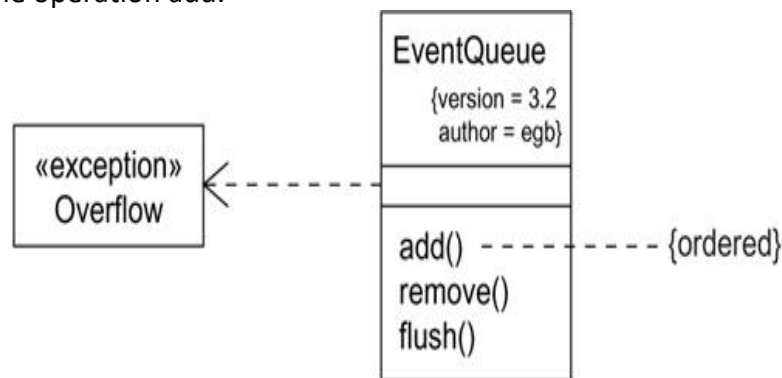
A *tagged value* extends the properties of a UML building block, allowing you to create new information in that element's specification. For example, if you want to specify the version and author of certain critical abstractions. Version and author are not primitive UML concepts.

- For example, the class EventQueue is extended by marking its version and author explicitly.



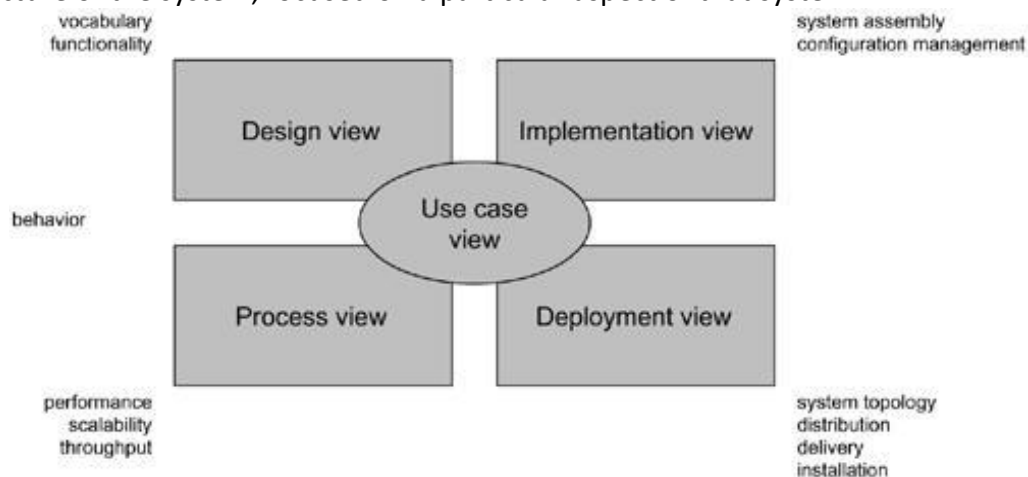
### 3. Constraints

A *constraint* extends the semantics of a UML building block, allowing you to add new rules or modify existing ones. For example, you might want to constrain the EventQueue class so that all additions are done in order. As Figure shows, you can add a constraint that explicitly marks these for the operation add.



## → Architecture:-

Software architecture is not only concerned with structure and behavior, but also with usage, functionality, performance, resilience, reuse, comprehensibility, economic and technology constraints. Below Figure shows, the architecture of a software-intensive system can best be described by five interlocking views. Each view is a projection into the organization and structure of the system, focused on a particular aspect of that system.



### 1. Design view:

The *design view* of a system encompasses the classes, interfaces, and collaborations that form the vocabulary of the problem and its solution. This view primarily supports the functional requirements of the system, meaning the services that the system should provide to its end users. With the UML, the static aspects of this view are captured in class diagrams and object diagrams; the dynamic aspects of this view are captured in interaction diagrams, statechart diagrams, and activity diagram.

## **2. Process view:**

The process view of a system encompasses the threads and processes that form the system's concurrency and synchronization mechanisms. This view primarily addresses the performance, scalability and throughput of the system. With the UML, the static aspects of this view are captured in class diagrams and object diagrams; the dynamic aspects of this view are captured in interaction diagrams, statechart diagrams, and activity diagram.

## **3. Use case view :**

The *use case view* of a system encompasses the use cases that describe the behavior of the system as seen by its end users, analysts, and testers. With the UML, the static aspects of this view are captured in use case diagrams; the dynamic aspects of this view are captured in interaction diagrams, statechart diagrams, and activity diagrams.

## **4. Implementation View:**

The Implementation View of a system encompasses the components and files that are used to assemble and release the physical system. This view primarily addresses the configuration management of the system. With the UML, the static aspects of this view are captured in component diagrams; the dynamic aspects of this view are captured in interaction diagrams, statechart diagrams, and activity diagrams.

## **5. Deployment view:**

The *deployment view* of a system encompasses the nodes that form the system's hardware topology on which the system executes. This view primarily addresses the distribution, delivery, and installation of the parts that make up the physical system. With the UML, the static aspects of this view are captured in deployment diagrams; the dynamic aspects of this view are captured in interaction diagrams, statechart diagrams, and activity diagrams.

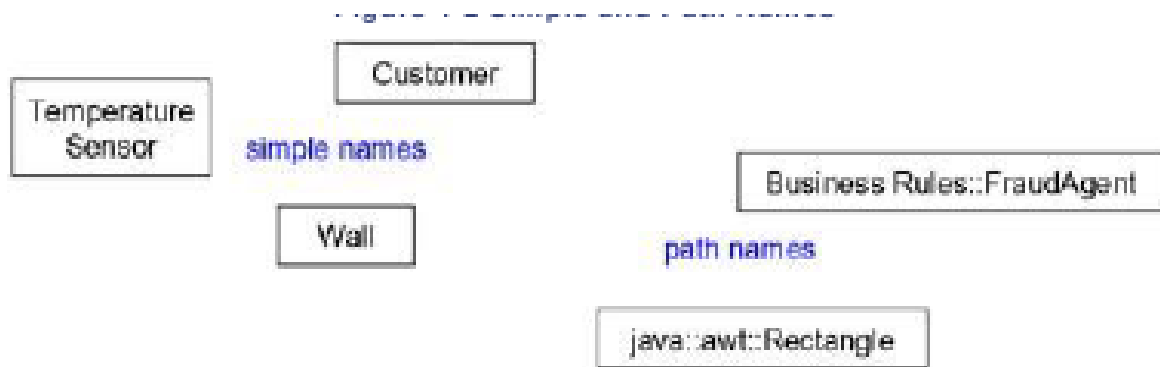
# → **Classes:-**

## **Terms and Concepts**

A class is a description of a set of objects that share the same attributes, operations, relationships, and semantics. Graphically, a class is rendered as a rectangle

## **Names:**

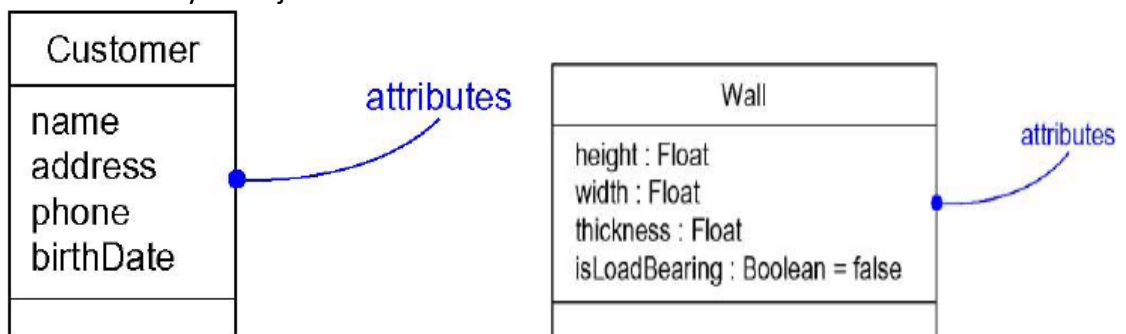
Every class must have a name that distinguishes it from other classes.. That name alone is known as a simple name; a path name is the class name prefixed by the name of the package in which that class lives. A class may be drawn showing only its name, as shown in Figure.



Class name may be text consisting of any number of letters, numbers, and certain punctuation marks (except for marks such as the colon, which is used to separate a class name and the name of its enclosing package) and may continue over several lines.

### **Attributes:**

An attribute is a named property of a class that describes a range of values that instances of the property may hold. A class may have any number of attributes or no attributes at all. An attribute represents some property of the thing we are modeling that is shared by all objects of that class.

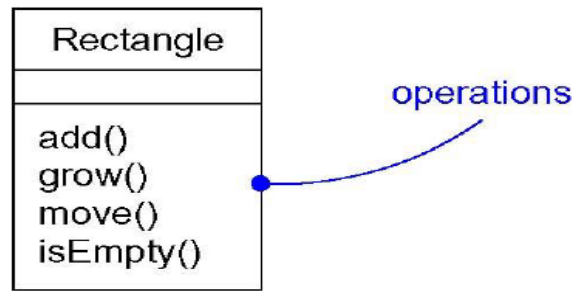


- For example, every wall has a height, width, and thickness; we might model our customers in such a way that each has a name, address, phone number, and date of birth.
- Graphically, attributes are listed in a compartment just below the class name. Attributes may be drawn showing only their names.

### **Operations**

An operation is the implementation of a service that can be requested from any object of the class to affect behavior. A class may have any number of operations or no operations at all.

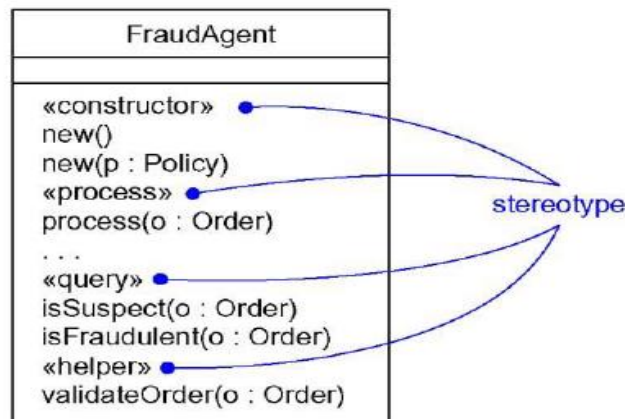




For example, all objects of the class **Rectangle** can be moved, resized, or queried for their properties. Graphically, operations are listed in a compartment just below the class attributes. Operations may be drawn showing only their names, as in Figure.

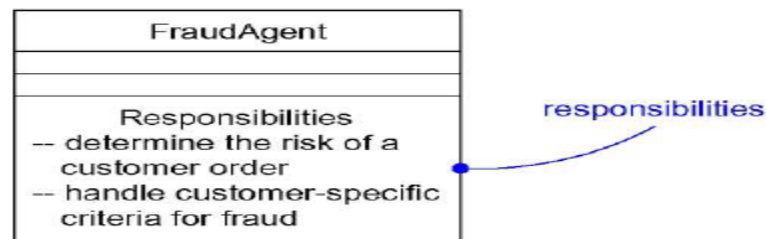
### Organizing Attributes and Operations

To better organize long lists of attributes and operations, we can also prefix each group with a descriptive category by using stereotypes, as shown in Figure



### Responsibilities

A responsibility is a contract or an obligation of a class. When we create a class, we are making a statement that all objects of that class have the same kind of state and the same kind of behavior. Graphically, responsibilities can be drawn in a separate compartment at the bottom of the class icon, as shown in Figure.



### Other Features

- Attributes, operations, and responsibilities are the most common features you'll need when we create abstractions. In fact, for most models we build, the basic form of these three features will be all we need to convey the most important semantics of your classes.

- Sometimes, we'll need to visualize or specify other features, such as the visibility of individual attributes and operations; language-specific features of an operation, such as whether it is polymorphic or constant; or even the exceptions that objects of the class might produce or handle.
- These and many other features can be expressed in the UML, but they are treated as advanced concepts.
- Finally, classes rarely stand alone. Rather, when we build models, we will typically focus on groups of classes that interact with one another.
- In the UML, these societies of classes form collaborations and are usually visualized in class diagrams.

## **Common Modeling Techniques of classes:**

### **1. Modeling the Vocabulary of a System**

To model the vocabulary of a system,

- Identify those things that users or implementers use to describe the problem or solution.
- For each abstraction, identify a set of responsibilities. Make sure that each class is crisply defined and that there is a good balance of responsibilities among all your classes.
- Provide the attributes and operations that are needed to carry out these responsibilities for each class.

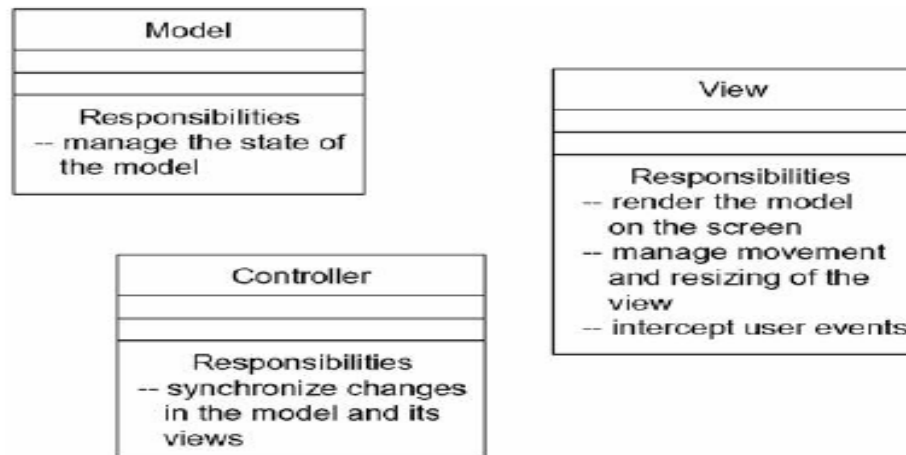


Figure shows a set of classes drawn from a retail system, including Customer, Order, and Product. This figure includes a few other related abstractions drawn from the vocabulary of the problem, such as Shipment (used to track orders), Invoice (used to bill orders), and Warehouse (where products are located prior to shipment). There is also one solution-related abstraction, Transaction, which applies to orders and shipments.

### **2. Modeling the Distribution of Responsibilities in a System**

To model the distribution of responsibilities in a system,

- Identify a set of classes that work together closely to carry out some behavior.
- Identify a set of responsibilities for each of these classes.
- Look at this set of classes as a whole, split classes that have too many responsibilities into smaller abstractions.

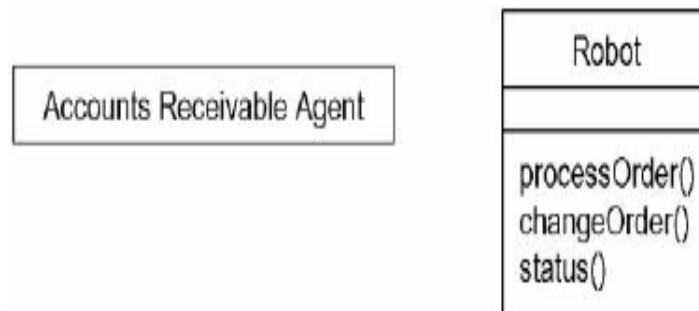


For example, Figure shows a set of classes drawn from Smalltalk, showing the distribution of responsibilities among Model, View, and Controller classes. Notice how all these classes work together such that no one class does too much or too little.

### **3. Modeling Nonsoftware Things**

To model nonsoftware things,

- Model the thing we are abstracting as a class.
- If we want to distinguish these things from the UML's defined building blocks, create a new building block by using stereotypes to specify these new semantics and to give a distinctive visual cue.
- If the thing we are modeling is some kind of hardware that itself contains software, consider modeling it as a kind of node, as well, so that we can further expand on its structure.

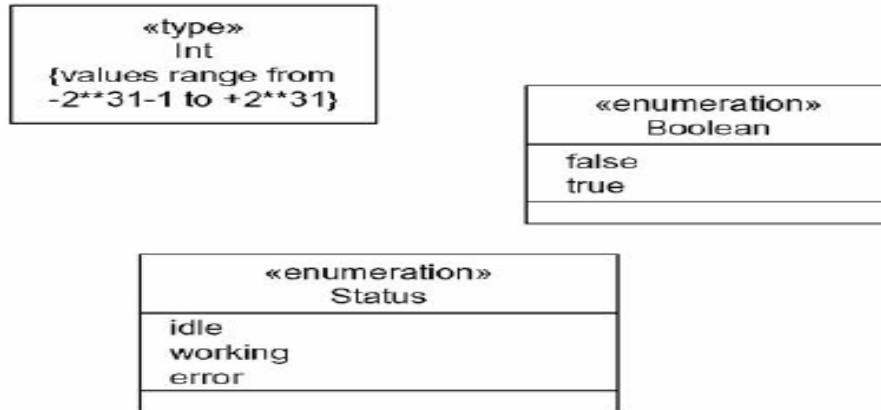


As Figure shows, it's perfectly normal to abstract humans (like AccountsReceivableAgent) and hardware (like Robot) as classes, because each represents a set of objects with a common structure and a common behavior.

### **4. Modeling Primitive Types**

To model primitive types,

- Model the thing we are abstracting as a type or an enumeration, which is rendered using class notation with the appropriate stereotype.
- If we need to specify the range of values associated with this type, use constraints.



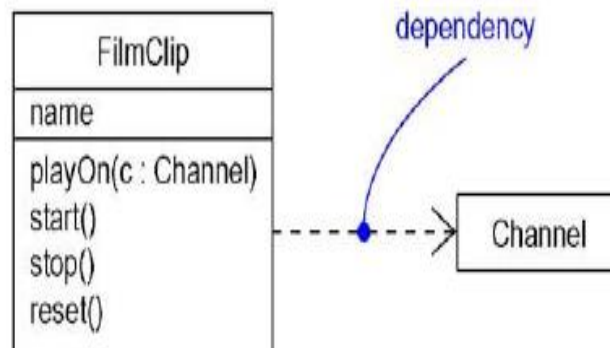
As Figure shows, these things can be modeled in the UML as types or enumerations, which are rendered just like classes but are explicitly marked via stereotypes. Things like integers (represented by the class Int) are modeled as types, and we can explicitly indicate the range of values these things can take on by using a constraint. Similarly, enumeration types, such as Boolean and Status, can be modeled as enumerations, with their individual values provided as attributes.

## → Relationships:

A **relationship** is a connection among things. In object-oriented modeling, the three most important relationships are dependencies, generalizations, and associations. Graphically, a relationship is rendered as a path, with different kinds of lines used to distinguish the kinds of relationships.

### 1. Dependency

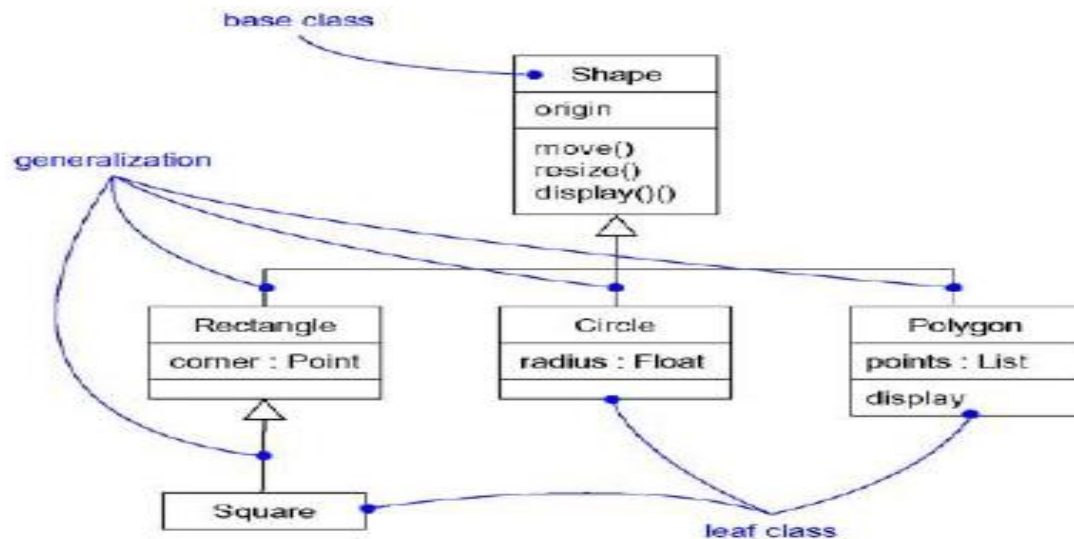
A dependency is a semantic relationship between two things in which a change to one thing (the independent thing) may affect the semantics of the other thing (the dependent thing). Graphically, a dependency is rendered as a dashed line, possibly directed, and occasionally including a label.



### 2. Generalization

A Generalization is a specialization/generalization relationship in which objects of the specialized element (the child) are substitutable for objects of the generalized element (the parent). In this way, the child shares the structure and the behavior of the parent. Graphically, a generalization relationship is rendered as a solid line with a hollow arrowhead pointing to the parent.

Generalization is sometimes called an "is-a-kind-of" relationship: one thing (like the class BayWindow) is-a-kind-of a more general thing (for example, the class Window).



- A class that has exactly one parent is said to use single inheritance; a class with more than one parent is said to use multiple inheritance.

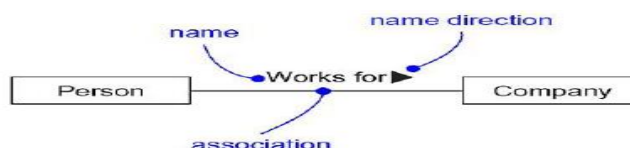
### 3.Association

An **association** is a structural relationship that describes a set of links, a link being a connection among objects. Graphically, an association is rendered as a solid line, possibly directed, occasionally including a label, and often containing other adornments, such as multiplicity and role names.



### Name

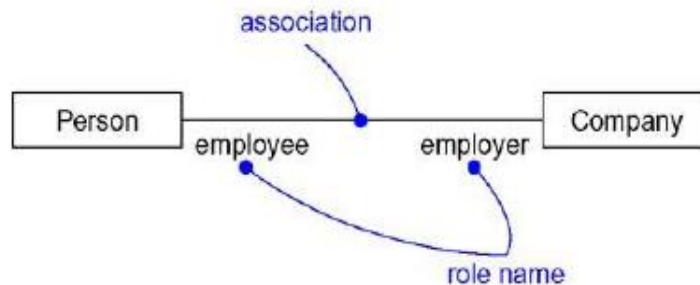
An association can have a name, and you use that name to describe the nature of the relationship. So that there is no ambiguity about its meaning, you can give a direction to the name by providing a direction triangle that point in the direction you intend to read the name, as shown in Figure.



## Role

When a class participates in an association, it has a specific role that it plays in that relationship;

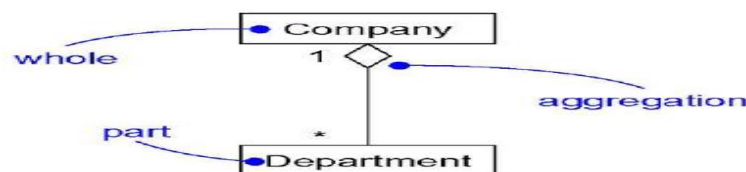
A role is just the face the class at the near end of the association presents to the class at the other end of the association.



In Figure, a Person playing the role of employee is associated with a Company playing the role of employer. An instance of an association is called a link. The same class can play the same or different roles in other associations

## Aggregation

Aggregation is a special kind of association, representing a structural relationship between a whole and its parts.



## Other Features

- Plain, unadorned dependencies, generalizations, and associations with names, multiplicities, and roles are the most common features you'll need when creating abstractions. In fact, for most of the models you build, the basic form of these three relationships will be all you need to convey the most important semantics of your relationships.
- Sometimes, you'll need to visualize or specify other features, such as composite aggregation, navigation, discriminates, association classes, and special kinds of dependencies and generalizations.
- These and many other features can be expressed in the UML, but they are treated as advanced concepts.
- Dependencies, generalization, and associations are all static things defined at the level of classes.
- In the UML, these relationships are usually visualized in class diagrams.
- When you start modeling at the object level, and especially when you start working with dynamic collaborations of these objects, you'll encounter two other kinds of relationships, links (which are instances of associations representing connections among

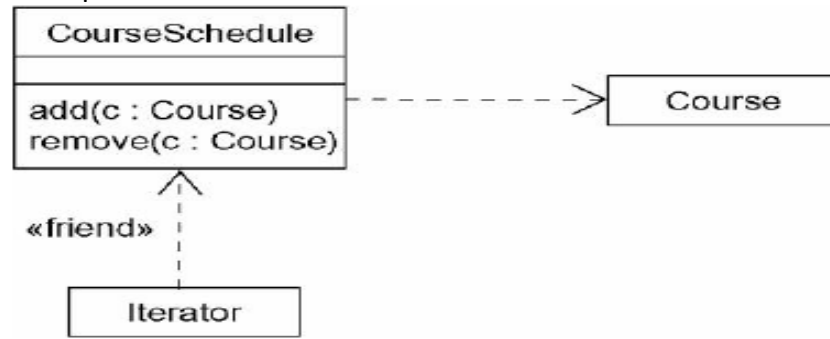
objects across which messages may be sent) and transitions (which are connections among states in a state machine).

## **Common Modeling Techniques of classes:**

### **Modeling Simple Dependencies**

To model this using relationship,

- Create a dependency pointing from the class with the operation to the class used as a parameter in the operation.



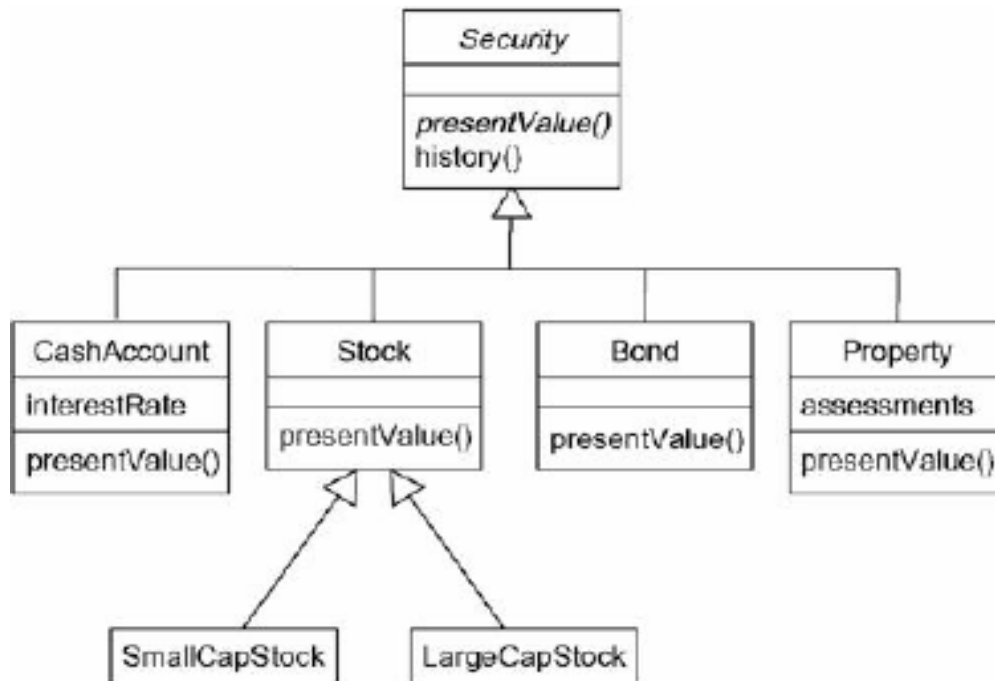
For example, Figure shows a set of classes drawn from a system that manages the assignment of students and instructors to courses in a university. This figure shows a dependency from CourseSchedule to Course, because Course is used in both the add and remove operations of CourseSchedule.

The dependency from Iterator shows that the Iterator uses the CourseSchedule; the CourseSchedule knows nothing about the Iterator. The dependency is marked with a stereotype, which specifies that this is not a plain dependency, but, rather, it represents a friend, as in C++.

### **Modeling Single Inheritance**

To model inheritance relationships,

- Given a set of classes, look for responsibilities, attributes, and operations that are common to two or more classes.
- Elevate these common responsibilities, attributes, and operations to a more general class. If necessary, create a new class to which you can assign these elements (but be careful about introducing too many levels).
- Specify that the more-specific classes inherit from the more-general class by placing a generalization relationship that is drawn from each specialized class to its more-general parent.



For example, Figure shows a set of classes drawn from a trading application. You will find a generalization relationship from four classes• CashAccount, Stock, Bond, and Property, to the more-general class named Security. Security is the parent, and CashAccount, Stock, Bond, and Property are all children. Each of these specialized children is a kind of Security. You'll notice that Security includes two operations: presentValue and history. Because Security is their parent, CashAccount, Stock, Bond, and Property all inherit these two operations, and for that matter, any other attributes and operations of Security that may be elided in this figure.

You can also create classes that have more than one parent. This is called multiple inheritance and means that the given class has all the attributes, operations, and associations of all its parents.

### **Modeling Structural Relationships:**

To model structural relationships,

- For each pair of classes, if you need to navigate from objects of one to objects of another, specify an association between the two. This is a data-driven view of associations.
- For each pair of classes, if objects of one class need to interact with objects of the other class other than as parameters to an operation, specify an association between the two. This is more of a behavior-driven view of associations.
- For each of these associations, specify a multiplicity (especially when the multiplicity is not \*, which is the default), as well as role names (especially if it helps to explain the model).
- If one of the classes in an association is structurally or organizationally a whole compared with the classes at the other end that look like parts, mark this as an aggregation by adorning the association at the end near the whole.



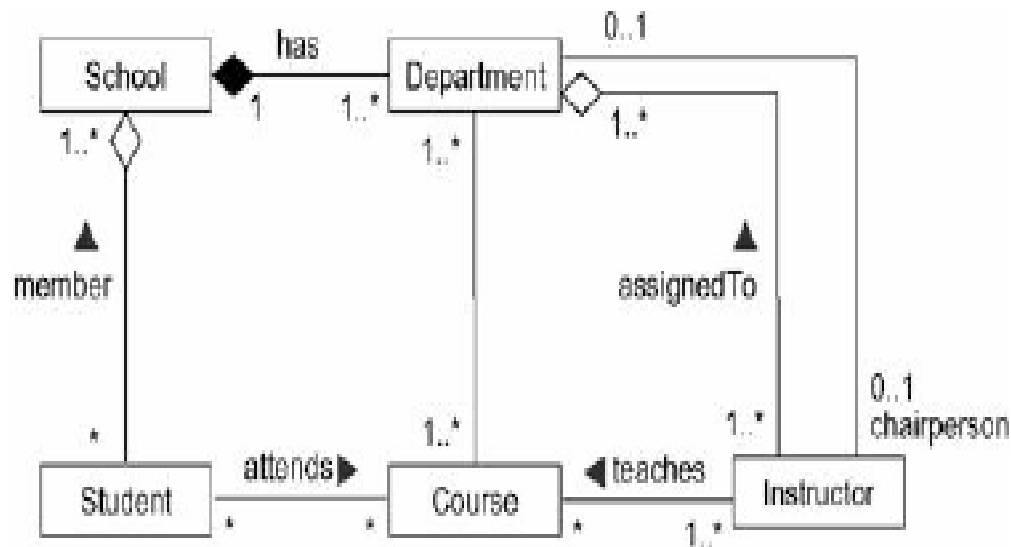


Figure shows a set of classes drawn from an information system for a school. The aggregation relationship between **School** and **Department** is composite aggregation. The relationships between **School** and the classes **Student** and **Department** are a bit different. Here you'll see aggregation relationships. A school has zero or more students; each student may be a registered member of one or more schools. A school has one or more departments; each department belongs to exactly one school. You could leave off the aggregation adornments and use plain associations, but by specifying that **School** is a whole and that **Student** and **Department** are some of its parts, you make clear which one is organizationally superior to the other. Thus, schools are somewhat defined by the students and departments they have. Similarly, students and departments don't really stand alone outside the school to which they belong. Rather, they get some of their identity from their school.

You'll also see that there are two associations between **Department** and **Instructor**. One of these associations specifies that every instructor is assigned to one or more departments and that each department has one or more instructors. This is modeled as an aggregation because organizationally, departments are at a higher level in the school's structure than are instructors. The other association specifies that for every department, there is exactly one instructor who is the department chair. The way this model is specified, an instructor can be the chair of no more than one department and some instructors are not chairs of any department.

## → Common Mechanisms

UML is made simpler by the presence of four common mechanisms that apply consistently throughout the language. They are:

1. Specifications
2. Adornments
3. Common divisions
4. Extensibility mechanisms

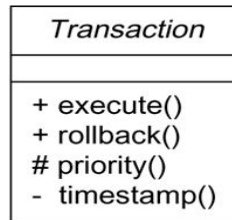
### 1. Specifications

The UML is more than just a graphical language. Rather, behind every part of its graphical notation there is a specification that provides a textual statement of the syntax and semantics of that building block. For example, behind a class icon is a specification that

provides the full set of attributes, operations (including their full signatures), and behaviors that the class embodies;

## **2.Adornments**

Most elements in the UML have a unique and direct graphical notation that provides a visual representation of the most important aspects of the element. A class's specification may include other details, such as whether it is abstract or the visibility of its attributes and operations. Many of these details can be rendered as graphical or textual adornments to the class's basic rectangular notation.



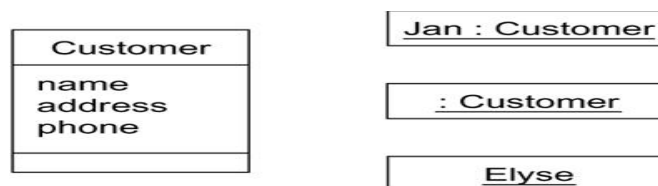
For example, Figure shows a class, adorned to indicate that it is an abstract class with two public, one protected, and one private operation. Every element in the UML's notation starts with a basic symbol, to which can be added a variety of adornments specific to that symbol.

## **3.Common Divisions**

In modeling object-oriented systems, the world often gets divided in at least a couple of ways.

### **1. class and object:**

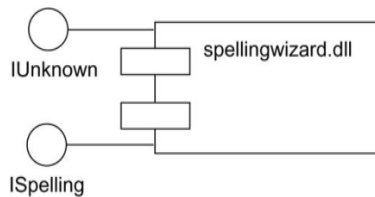
A class is an abstraction; an object is one concrete manifestation of that abstraction. In the UML, you can model classes as well as objects, as shown in Figure



In this figure, there is one class, named *Customer*, together with three objects: *Jan* (which is marked explicitly as being a *Customer* object), *:Customer* (an anonymous *Customer* object), and *Elyse* (which in its specification is marked as being a kind of *Customer* object, although it's not shown explicitly here). Graphically, the UML distinguishes an object by using the same symbol as its class and then simply underlining the object's name.

### **2. interface and implementation.:**

An interface declares a contract, and an implementation represents one concrete realization of that contract, responsible for faithfully carrying out the interface's complete semantics. In the UML, you can model both interfaces and their implementations, as shown in Figure



In this figure, there is one component named spellingwizard.dll that implements two interfaces, IUnknown and ISpelling.

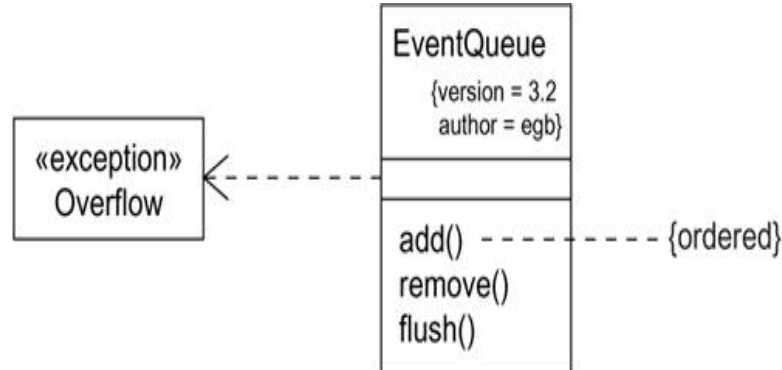
#### **4.Extensibility Mechanisms**

The UML provides a standard language for writing software blueprints, but it is not possible for one closed language to ever be sufficient to express all possible nuances of all models across all domains across all time. The UML's extensibility mechanisms include

1. Stereotypes
2. Tagged values
3. Constraints

##### **1. Stereotypes**

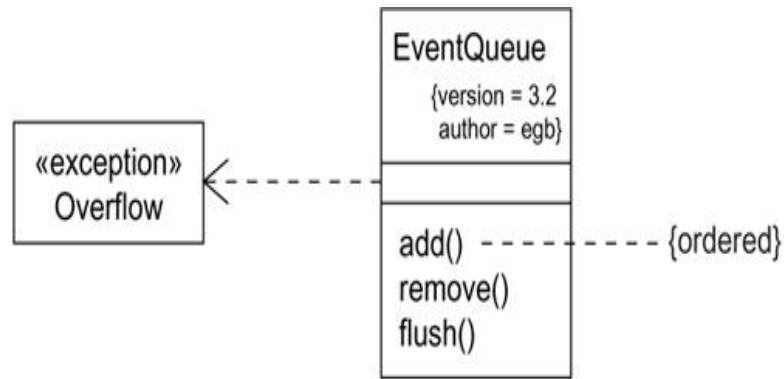
A *stereotype* extends the vocabulary of the UML, allowing you to create new kinds of *building* blocks that are derived from existing ones but that are specific to your problem. For example, if you are working in a programming language, such as Java or C++, you will often want to model exceptions. In these languages, exceptions are just classes. You can make exceptions first class citizens in your models, meaning that they are treated like basic building blocks, by marking them with an appropriate stereotype, as for the class Overflow in Figure.



##### **2. Tagged values**

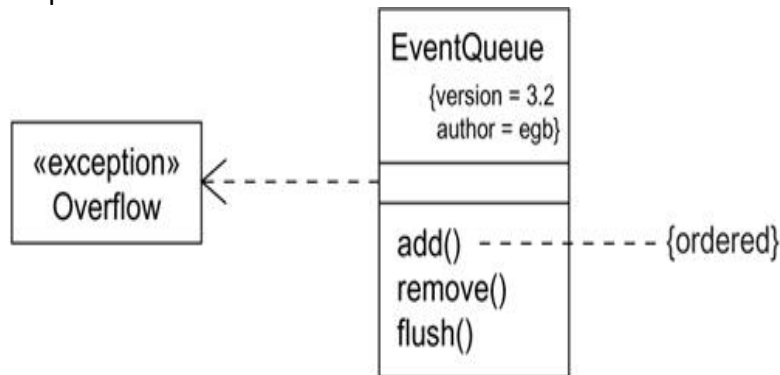
A *tagged value* extends the properties of a UML building block, allowing you to create new information in that element's specification. For example, if you want to specify the version and author of certain critical abstractions. Version and author are not primitive UML concepts.

- For example, the class EventQueue is extended by marking its version and author explicitly.



### 3. Constraints

A *constraint* extends the semantics of a UML building block, allowing you to add new rules or modify existing ones. For example, you might want to constrain the EventQueue class so that all additions are done in order. As Figure shows, you can add a constraint that explicitly marks these for the operation add.

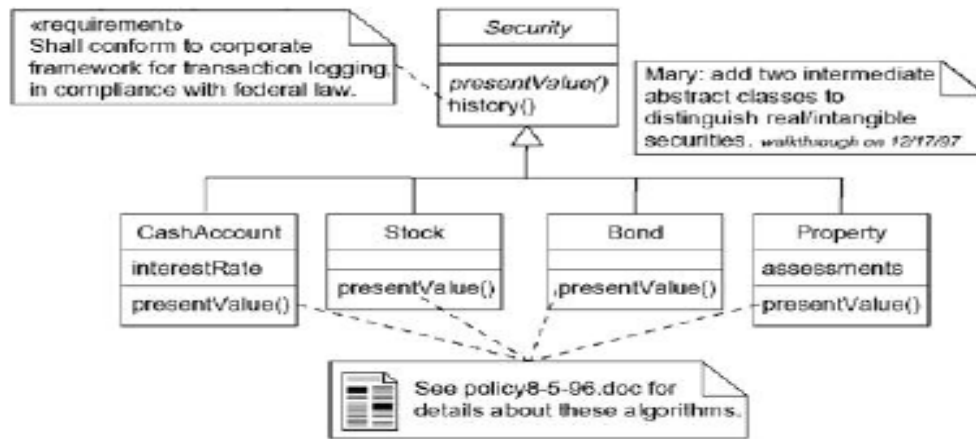


## Common Modeling Techniques of common mechanisms

### Modeling Comments

To model a comment,

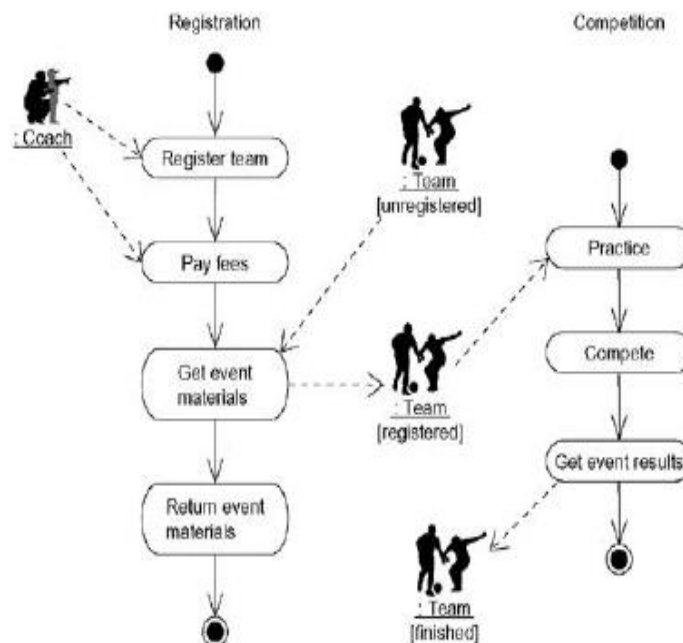
- Put your comment as text in a note and place it adjacent to the element to which it refers. You can show a more explicit relationship by connecting a note to its elements using a dependency relationship.
- Remember that you can hide or make visible the elements of your model as you see fit. This means that you don't have to make your comments visible everywhere the elements to which it is attached are visible. Rather, expose your comments in your diagrams only insofar as you need to communicate that information in that context.
- If your comment is lengthy or involves something richer than plain text, consider putting your comment in an external document and linking or embedding that document in a note attached to your model.
- As your model evolves, keep those comments that record significant decisions that cannot be inferred from the model itself, and unless they are of historic interest discard the others.



## Modeling New Building Blocks .

To model new building blocks,

- Make sure there's not already a way to express what you want by using basic UML. If you have a common modeling problem, chances are there's already some standard stereotype that will do what you want.
- If you're convinced there's no other way to express these semantics, identify the primitive thing in the UML that's most like what you want to model (for example, class, interface, component, node, association, and so on) and define a new stereotype for that thing. Remember that you can define hierarchies of stereotypes so that you can have general kinds of stereotypes along with their specializations (but as with any hierarchy, use this sparingly).
- Specify the common properties and semantics that go beyond the basic element being stereotyped by defining a set of tagged values and constraints for the stereotype.
- If you want these stereotype elements to have a distinctive visual cue, define a new icon for the stereotype.

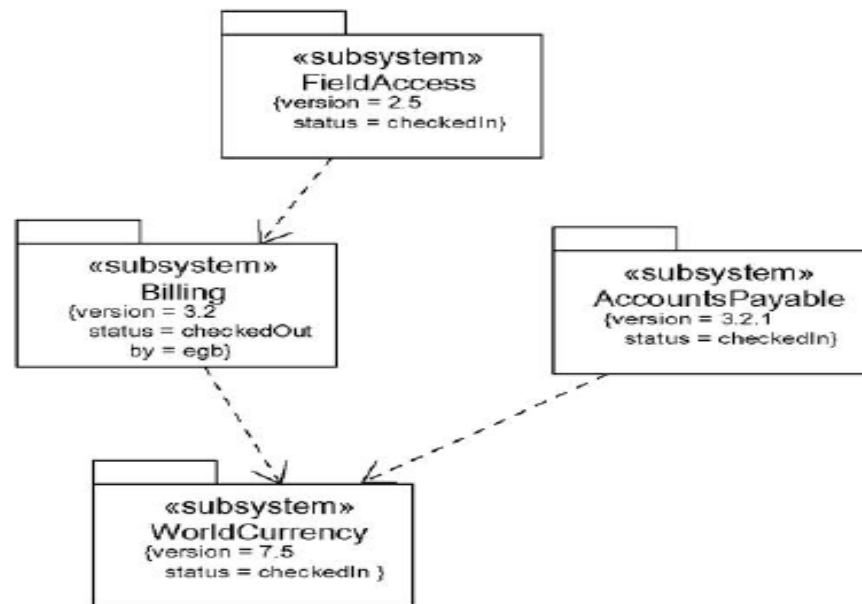


For example, suppose you are using activity diagrams to model a business process involving the flow of coaches and teams through a sporting event. In this context, it would make sense to visually distinguish coaches and teams from one another and from the other things in this domain, such as events and divisions.

## Modeling New Properties

To model new properties,

- First, make sure there's not already a way to express what you want by using basic UML. If you have a common modeling problem, chances are that there's already some standard tagged value that will do what you want.
- If you're convinced there's no other way to express these semantics, add this new property to an individual element or a stereotype. The rules of generalization apply•tagged values defined for one kind of element apply to its children.



### **Note**

The values of tags such as *version* and *status* are things that can be set by tools. Rather than setting these values in your model by hand, you can use a development environment that integrates your configuration management tools with your modeling tools to maintain these values for you.

For example, suppose you want to tie the models you create to your project's configuration management system. Among other things, this means keeping track of the version number, current check in/check out status, and perhaps even the creation and modification dates of each subsystem. Because this is process-specific information, it is not a basic part of the UML, although you can add this information as tagged values. Furthermore, this information is not just a class attribute either. A subsystem's version number is part of its metadata, not part of the model.

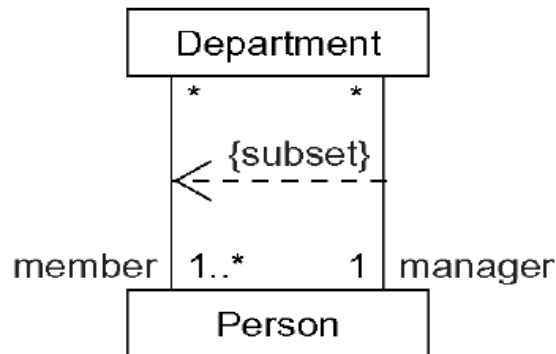
Figure shows four subsystems, each of which has been extended to include its version number and status. In the case of the Billing subsystem, one other tagged value is shown the person who has currently checked out the subsystem.

## Modeling New Semantics

To model new semantics,

- First, make sure there's not already a way to express what you want by using basic UML. If you have a common modeling problem, chances are that there's already some standard constraint that will do what you want.

- If you're convinced there's no other way to express these semantics, write your new semantics as text in a constraint and place it adjacent to the element to which it refers. You can show a more explicit relationship by connecting a constraint to its elements using a dependency relationship.
- If you need to specify your semantics more precisely and formally, write your new semantics using OCL.



For example, Figure models a small part of a corporate human resources system. This diagram shows that each Person may be a member of zero or more Departments and that each Department must have at least one Person as a member. This diagram goes on to indicate that each Department must have exactly one Person as a manager and every Person may be the manager of zero or more Departments. All of these semantics can be expressed using simple UML. However, to assert that a manager must also be a member of the department is something that cuts across multiple associations and cannot be expressed using simple UML. To state this invariant, you have to write a constraint that shows the manager as a subset of the members of the Department, connecting the two associations and the constraint by a dependency from the subset to the superset.

## → Class Diagrams :

Class diagrams are the most common diagram found in modeling object-oriented systems. A class diagram shows a set of classes, interfaces, and collaborations and their relationships. You use class diagrams to model the static design view of a system.

### Terms and Concepts :

A class diagram is a diagram that shows a set of classes, interfaces, and collaborations and their relationships. Graphically, a class diagram is a collection of vertices and arcs.

### Common Properties

A class diagram is just a special kind of diagram and shares the same common properties as do all other diagrams, a name and graphical content that are a projection into a model. What distinguishes a class diagram from all other kinds of diagrams is its particular content.

### Contents

Class diagrams commonly contain the following things:

- Classes
- Interfaces
- Collaborations
- Dependency, generalization, and association relationships .

Like all other diagrams, class diagrams may contain notes and constraints. Class diagrams may also contain packages or subsystems.

### **Note**

Component diagrams and deployment diagrams are similar to class diagrams, except that instead of containing classes, they contain components and nodes, respectively.

### **Common Uses**

You use class diagrams to model the static design view of a system. This view primarily supports the functional requirements of a system • the services the system should provide to its end users. When you model the static design view of a system, you'll typically use class diagrams in one of three ways.

#### **1. To model the vocabulary of a system**

Modeling the vocabulary of a system involves making a decision about which abstractions are a part of the system under consideration and which fall outside its boundaries. You use class diagrams to specify these abstractions and their responsibilities.

#### **2. To model simple collaborations**

A collaboration is a society of classes, interfaces, and other elements that work together to provide some cooperative behavior that's bigger than the sum of all the elements. For example, when you're modeling the semantics of a transaction in a distributed system, you can't just stare at a single class to understand what's going on. Rather, these semantics are carried out by a set of classes that work together. You use class diagrams to visualize and specify this set of classes and their relationships.

#### **3. To model a logical database schema**

Think of a schema as the blueprint for the conceptual design of a database. In many domains, you'll want to store persistent information in a relational database or in an object-oriented database. You can model schemas for these databases using class diagrams.

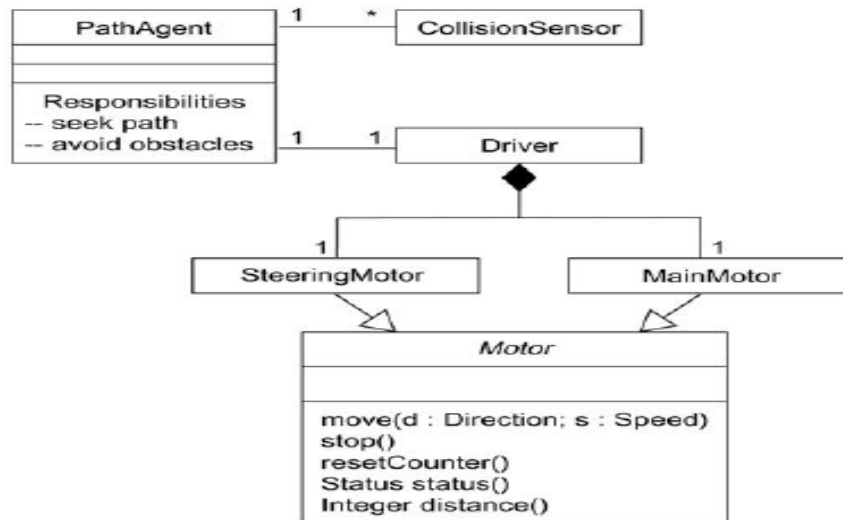
## **Common Modeling Techniques of class diagrams:**

### **1. Modeling Simple Collaborations**



To model a collaboration,

- Identify the mechanism you'd like to model. A mechanism represents some function or behavior of the part of the system you are modeling that results from the interaction of a society of classes, interfaces, and other things.
- For each mechanism, identify the classes, interfaces, and other collaborations that participate in this collaboration. Identify the relationships among these things, as well.
- Use scenarios to walk through these things. Along the way, you'll discover parts of your model that were missing and parts that were just plain semantically wrong.
- Be sure to populate these elements with their contents. For classes, start with getting a good balance of responsibilities. Then, over time, turn these into concrete attributes and operations.



For example, Figure shows a set of classes drawn from the implementation of an autonomous robot. The figure focuses on the classes involved in the mechanism for moving the robot along a path. You'll find one abstract class (Motor) with two concrete children, SteeringMotor and MainMotor. Both of these classes inherit the five operations of their parent, Motor. The two classes are, in turn, shown as parts of another class, Driver. The class PathAgent has a one-to-one association to Driver and a one-to-many association to CollisionSensor. No attributes or operations are shown for PathAgent, although its responsibilities are given.

## **Modeling a Logical Database Schema**

**To model a schema,**

- Identify those classes in your model whose state must transcend the lifetime of their applications.
- Create a class diagram that contains these classes and mark them as persistent (a standard tagged value). You can define your own set of tagged values to address database-specific details.
- Expand the structural details of these classes. In general, this means specifying the details of their attributes and focusing on the associations and their cardinalities that structure these classes.

- Watch for common patterns that complicate physical database design, such as cyclic associations, one-to-one associations, and n-ary associations. Where necessary, create intermediate abstractions to simplify your logical structure.
- Consider also the behavior of these classes by expanding operations that are important for data access and data integrity. In general, to provide a better separation of concerns, business rules concerned with the manipulation of sets of these objects should be encapsulated in a layer above these persistent classes.
- Where possible, use tools to help you transform your logical design into a physical design.

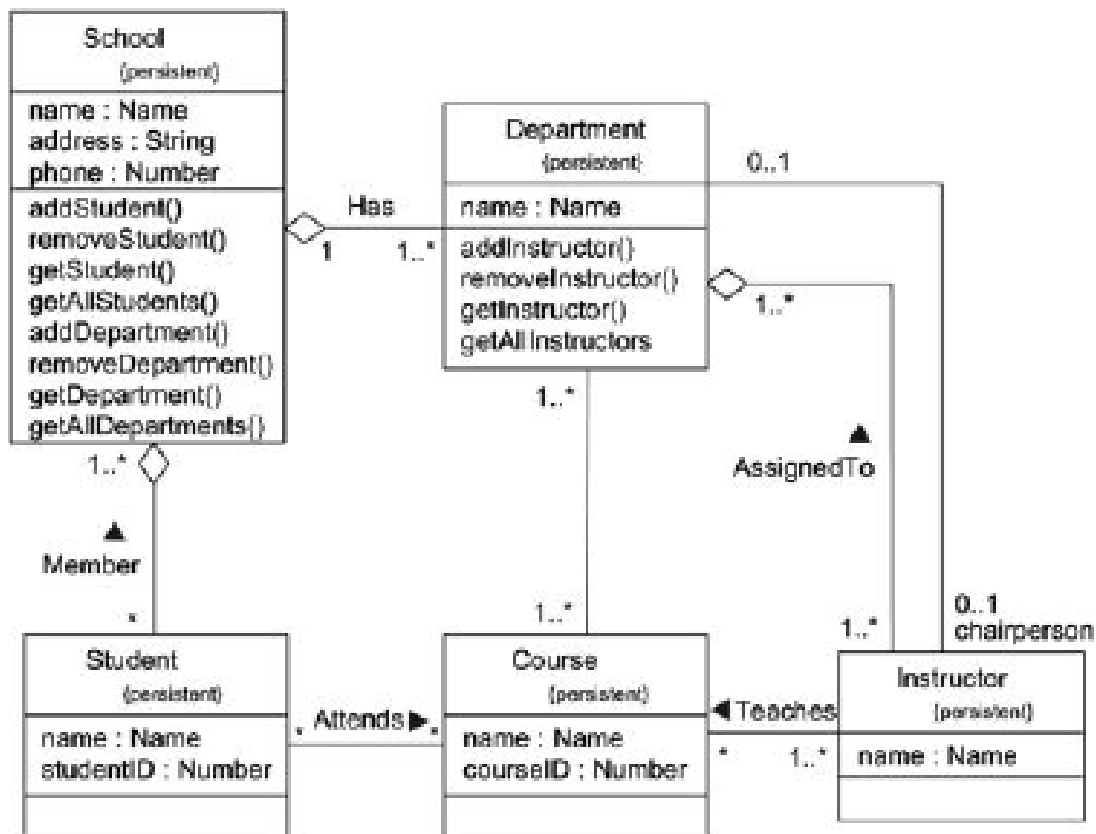


Figure shows a set of classes drawn from an information system for a school. This figure expands upon an earlier class diagram, and you'll see the details of these classes revealed to a level sufficient to construct a physical database. Starting at the bottom-left of this diagram, you will find the classes named Student, Course, and Instructor. There's an association between Student and Course, specifying that students attend courses. Furthermore, every student may attend any number of courses and every course may have any number of students.

All of these classes are marked as persistent, indicating that their instances are intended to live in a database or some other form of persistent store. This diagram also exposes the attributes of all six of these classes. Notice that all the attributes are primitive types. When you are modeling a schema, you'll generally want to model the relationship to any nonprimitive types using an explicit aggregation rather than an attribute.

## Forward and Reverse Engineering

Forward engineering is the process of transforming a model into code through a mapping to an implementation language. Forward engineering results in a loss of information, because models written in the UML are semantically richer than any current object-oriented programming language.

To forward engineer a class diagram,

- Identify the rules for mapping to your implementation language or languages of choice. This is something you'll want to do for your project or your organization as a whole.
- Depending on the semantics of the languages you choose, you may have to constrain your use of certain UML features. For example, the UML permits you to model multiple inheritance, but Smalltalk permits only single inheritance. You can either choose to prohibit developers from modeling with multiple inheritance (which makes your models language-dependent) or develop idioms that transform these richer features into the implementation language (which makes the mapping more complex).
- Use tagged values to specify your target language. You can do this at the level of individual classes if you need precise control. You can also do so at a higher level, such as with collaborations or packages.
- Use tools to forward engineer your models.

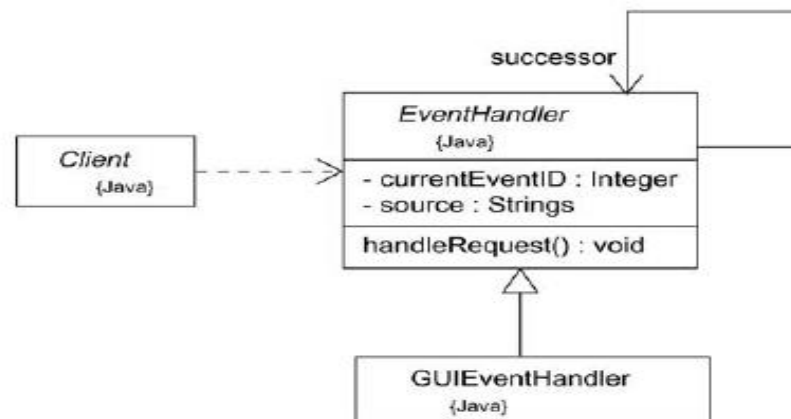


Figure illustrates a simple class diagram specifying an instantiation of the chain of responsibility pattern. This particular instantiation involves three classes: **Client**, **EventHandler**, and **GUIEventHandler**. **Client** and **EventHandler** are shown as abstract classes, whereas **GUIEventHandler** is concrete. **EventHandler** has the usual operation expected of this pattern (`handleRequest`), although two private attributes have been added for this instantiation.

All of these classes specify a mapping to Java, as noted in their tagged value. Forward engineering the classes in this diagram to Java is straightforward, using a tool. Forward engineering the class **EventHandler** yields the following code.

```
public abstract class EventHandler {

    EventHandler successor;
    private Integer currentEventID;
```

```
private String source;

EventHandler() {}
public void handleRequest() {}
}
```

Reverse engineering is the process of transforming code into a model through a mapping from a specific implementation language

To reverse engineer a class diagram,

- Identify the rules for mapping from your implementation language or languages of choice. This is something you'll want to do for your project or your organization as a whole.
- Using a tool, point to the code you'd like to reverse engineer. Use your tool to generate a new model or modify an existing one that was previously forward engineered.
- Using your tool, create a class diagram by querying the model. For example, you might start with one or more classes, then expand the diagram by following specific relationships or other neighboring classes. Expose or hide details of the contents of this class diagram as necessary to communicate your intent.

## → **Object Diagrams:**

### **Terms and Concepts**

An object diagram is a diagram that shows a set of objects and their relationships at a point in time. Graphically, an object diagram is a collection of vertices and arcs.

#### **Common Properties**

An object diagram is a special kind of diagram and shares the same common properties as all other diagrams, that is, a name and graphical contents that are a projection into a model. What distinguishes an object diagram from all other kinds of diagrams is its particular content.

#### **Contents**

Object diagrams commonly contain

- Objects
- Links

Like all other diagrams, object diagrams may contain notes and constraints. Object diagrams may also contain packages or subsystems

#### **Common Uses**

You use object diagrams to model the static design view or static process view of a system just as you do with class diagrams, but from the perspective of real or prototypical instances. This view primarily supports the functional requirements of a system• that is, the services the system should provide to its end users. Object diagrams let you model static data structures.

When you model the static design view or static process view of a system, you typically use object diagrams in one way:

- To model object structures

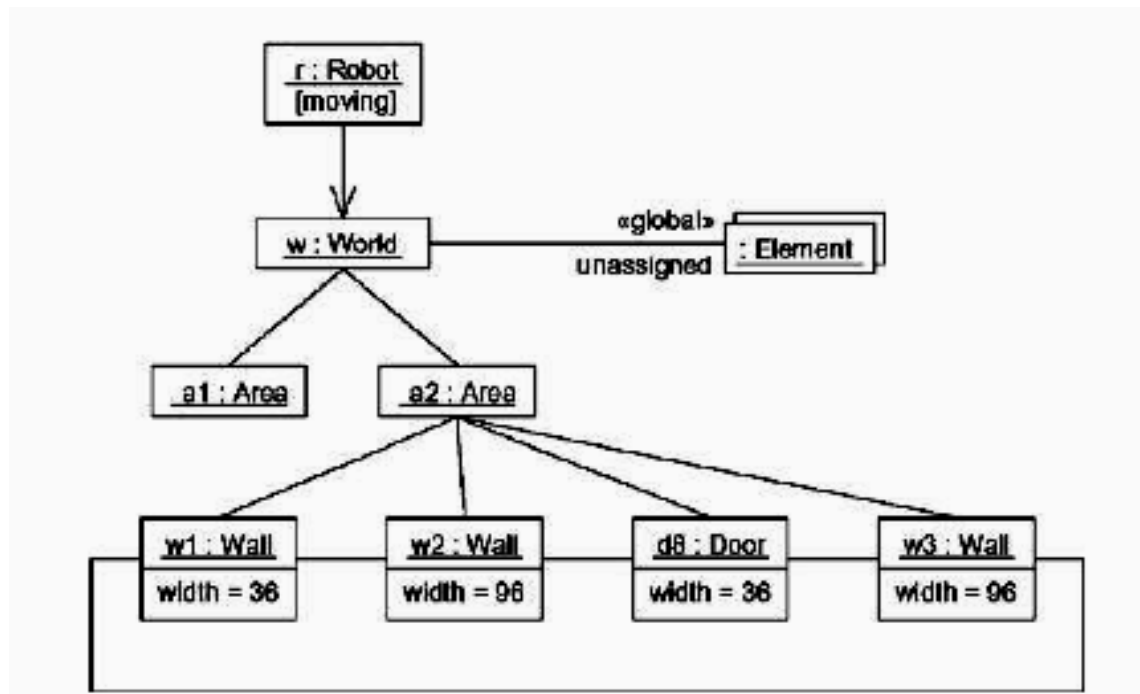
Modeling object structures involves taking a snapshot of the objects in a system at a given moment in time. An object diagram represents one static frame in the dynamic storyboard represented by an interaction diagram. You use object diagrams to visualize, specify, construct, and document the existence of certain instances in your system, together with their relationships to one another.

## **Common Modeling Techniques of object diagrams:**

### **Modeling Object Structures**

To model an object structure,

- Identify the mechanism you'd like to model. A mechanism represents some function or behavior of the part of the system you are modeling that results from the interaction of a society of classes, interfaces, and other things.
- For each mechanism, identify the classes, interfaces, and other elements that participate in this collaboration; identify the relationships among these things, as well.
- Consider one scenario that walks through this mechanism. Freeze that scenario at a moment in time, and render each object that participates in the mechanism.
- Expose the state and attribute values of each such object, as necessary, to understand the scenario.
- Similarly, expose the links among these objects, representing instances of associations among them.



For example, Figure shows a set of objects drawn from the implementation of an autonomous robot. This figure focuses on some of the objects involved in the mechanism used by the robot to calculate a model of the world in which it moves. There are many more objects involved in a running system, but this diagram focuses on only those abstractions that are directly involved in creating this world view.

As this figure indicates, one object represents the robot itself (*r*, an instance of *Robot*), and *r* is currently in the state marked moving. This object has a link to *w*, an instance of *World*, which represents an abstraction of the robot's world model. This object has a link to a multiobject that consists of instances of *Element*, which represent entities that the robot has identified but not yet assigned in its world view. These elements are marked as part of the robot's global state.

At this moment in time, *w* is linked to two instances of *Area*. One of them (*a2*) is shown with its own links to three *Wall* and one *Door* object. Each of these walls is marked with its current width, and each is shown linked to its neighboring walls. As this object diagram suggests, the robot has recognized this enclosed area, which has walls on three sides and a door on the fourth.

### **Forward and Reverse Engineering**

Forward engineering (the creation of code from a model) an object diagram is theoretically possible but pragmatically of limited value. In an object-oriented system, instances are things that are created and destroyed by the application during run time. Therefore, you can't exactly instantiate these objects from the outside.

Reverse engineering (the creation of a model from code) an object diagram is a very different thing. In fact, while you are debugging your system, this is something that you or your tools will do all the time. For example, if you are chasing down a dangling link, you'll want to

literally or mentally draw an object diagram of the affected objects to see where, at a given moment in time, an object's state or its relationship to other objects is broken.

### **To reverse engineer an object diagram,**

- Chose the target you want to reverse engineer. Typically, you'll set your context inside an operation or relative to an instance of one particular class.
- Using a tool or simply walking through a scenario, stop execution at a certain moment in time.
- Identify the set of interesting objects that collaborate in that context and render them in an object diagram.
- As necessary to understand their semantics, expose these object's states.
- As necessary to understand their semantics, identify the links that exist among these objects.

If your diagram ends up overly complicated, prune it by eliminating objects that are not germane to the questions about the scenario you need answered. If your diagram is too simplistic, expand the neighbors of certain interesting objects and expose each object's state more deeply.

## **Short Questions**

1. What does Unified mean?
2. List the scope of UML.
3. List the types of relationships.
4. Give graphical representation diagram for dependencies, associations, generalizations and realizations.
5. Write about Association.
6. Differentiate static models and dynamic models.
7. Write about importance of Public, Private and Protected in class.
8. What is an interface?
9. What is the need for modeling?
10. Compare different class diagrams.

## **Essay Questions**

1. Explain the modeling of system's architecture with diagram.

(or)

Demonstrate the working of UML architecture with neat diagram.

(or)

Describe about five interlocking views involved in the architecture of a software-intensive system modeling

2. Explain about common mechanisms in the UML.
3. Explain the Conceptual model of UML in detail.

4. What is the UML approach to software development life cycle? Briefly explain various building blocks of UML.

5. List out the common properties of object and class diagram and demonstrate the classes and objects in bank management system with neat sketch.

## UNIT-IV







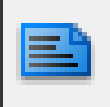



**STRUCTURAL MODELING:** Packages Diagram, Composite Structure Diagram, Component Diagram, Deployment Diagram, Profile Diagram.

---

### 1. Package Diagram:

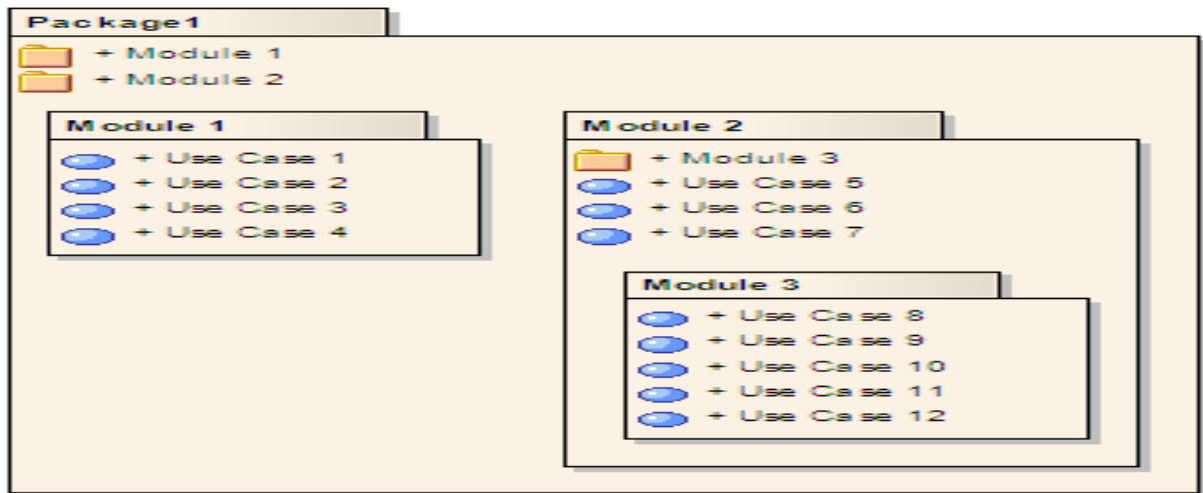
- A package diagram package diagram shows the dependencies between different packages in a system.
- Collection of vertices and arcs.
- Package diagram can also show both structure and dependencies between sub-systems or modules.

### Essential elements of Package diagram:

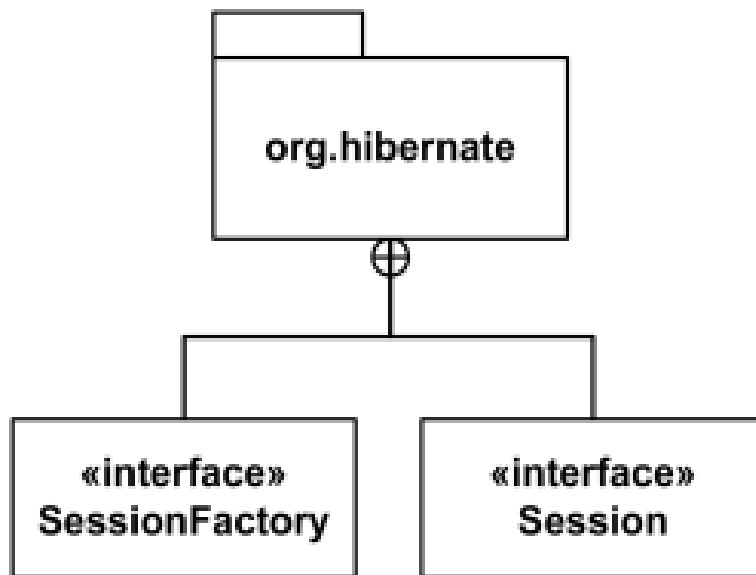
	Access		Constraint
	Dependency		Generalization
	Import		Merge
	Note		Package
	Realization		Subsystem

**Package:**





- A **package** is used to group elements, and provides a namespace for the grouped elements.
- A package may contain other packages.
- Owned members :
- Package can also be merge with other package .
- Different types of elements are allow to have the same name.
- Members of the package may be shown **outside** of the package by branching lines.
- Package org.hibernate contains interfaces Session and Session Factory.



### Visibility:

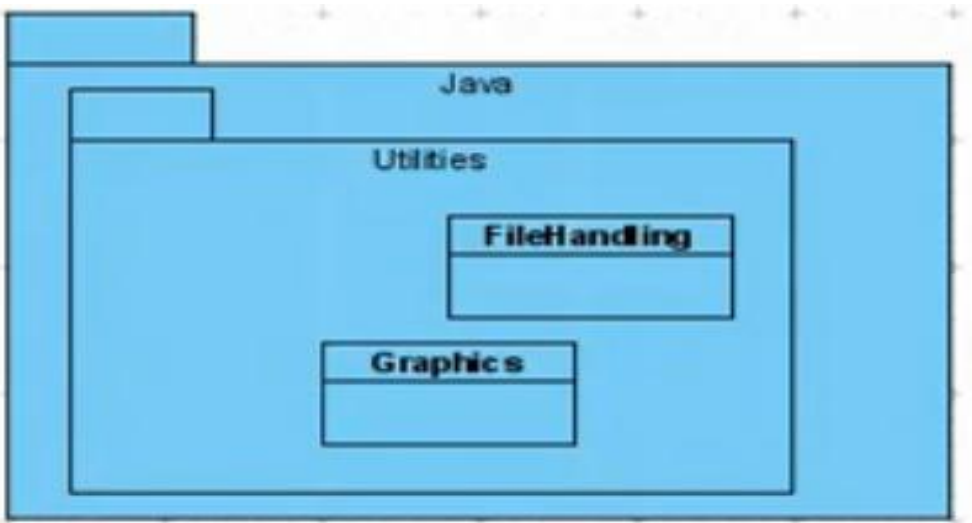
- **Visibility** of Owned element.
- "+" for public and "-" for private.
- All elements of Library Domain package are public except for Account.
-

## Library Domain

- + Catalog
- + Patron
- + Librarian
- Account

### Sub system :

- Packages are useful for simplify this kind of diagrams
- Nested packages.
- Qualifier for Graphics class is Java::Utilities::Graphics



- The elements that can be referred to within a package using **non-qualified** names are: Owned Element, Imported Element.
- Owned and imported elements may have a **visibility** that determines whether they are available outside the package.
- The elements that can be referred to within a package using **non-qualified** names are: Owned Element, Imported Element.
- Owned and imported elements may have a **visibility** that determines whether they are available outside the package.

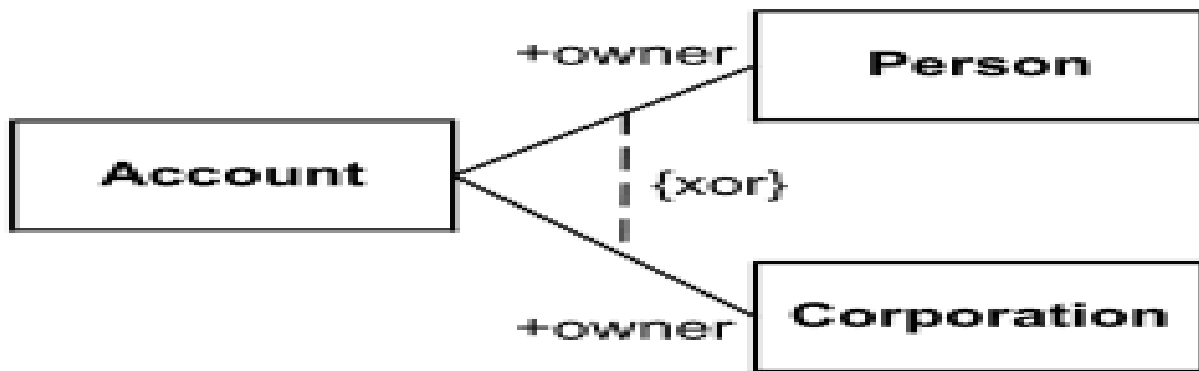
### NOTE:

- To provide explanation about any UML elements.
- Anchor note

## Note

### Constraint:

- A **constraint** is a [packageable element](#) which represents some **condition, restriction** related to several elements.
- usually specified by a Boolean expression
- { Constraint-name }
- may be placed near the symbol for the element



### Other Elements:

- **Access** : indicates only Owned elements of importing package are accessible.
- **Import** : It is a relationship between two packages that indicates the **contents of a package is imported by a other package.**
- **Merge**: It is a relationship between two packages that indicates the **contents of the two packages are to be combined.**

### Relationships:

- Dependency
- Generalization

### Dependency:

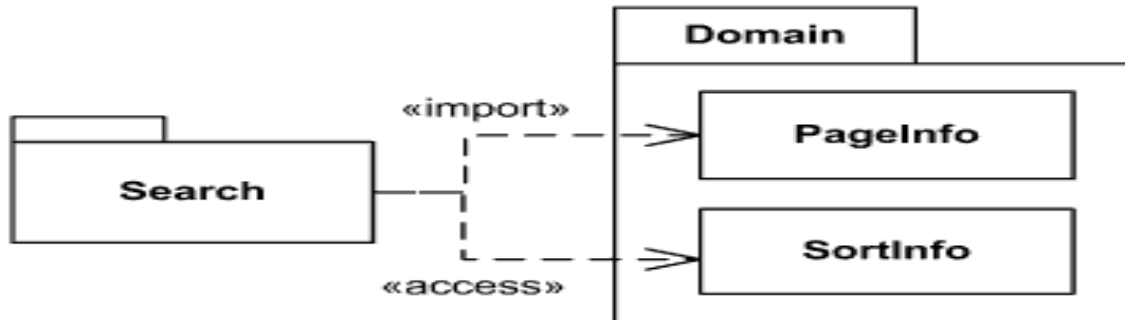
- **One Package depends on another package.**

### Refinement:

- Refinement shows different kind of relationship between packages.
- One Package refines another package, if it contains same elements but offers more details about those elements.

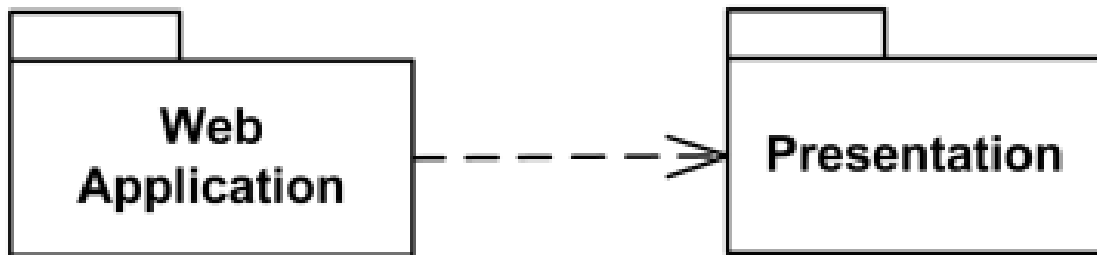
### Element Import:

- The keyword **«import»** - the visibility is **public**
- The keyword **«access»** -to indicate **private** visibility
- Public import of Page Info element and private import of Sort Info element from Domain package.

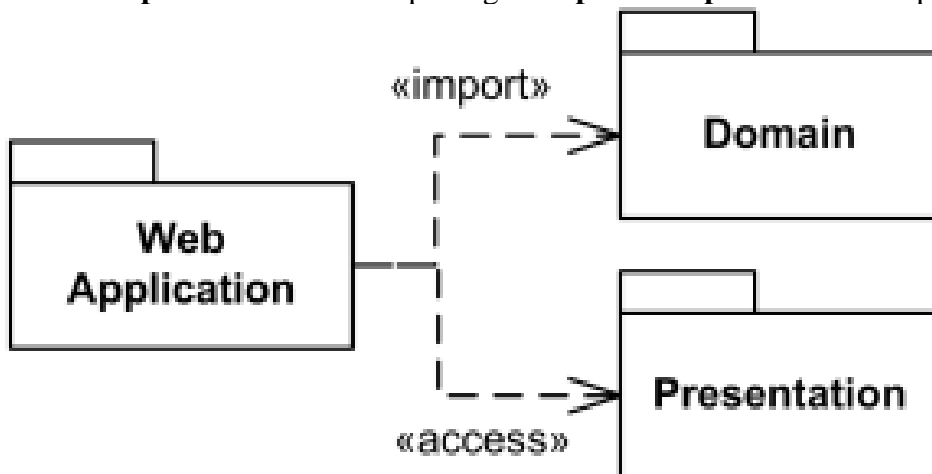


### Package Import:

- **Package Import** is a directed relationship between an importing **package** and imported **package**
- A package import is shown using a dashed arrow with an open arrowhead from the importing package to the imported package.
- It looks exactly like dependency and usage relationships.
- Web Application imports Presentation package.
- Importing: web application
- Imported: Presentation



- **Private import** of Presentation package and **public import** of Domain package.



## Package Merge:

- A package merge is a directed relationship between two packages.
- It indicates that content of one package is extended by the contents of another package.
- package merge is shown using a dashed line with an open arrowhead pointing from the receiving package to the merged package.

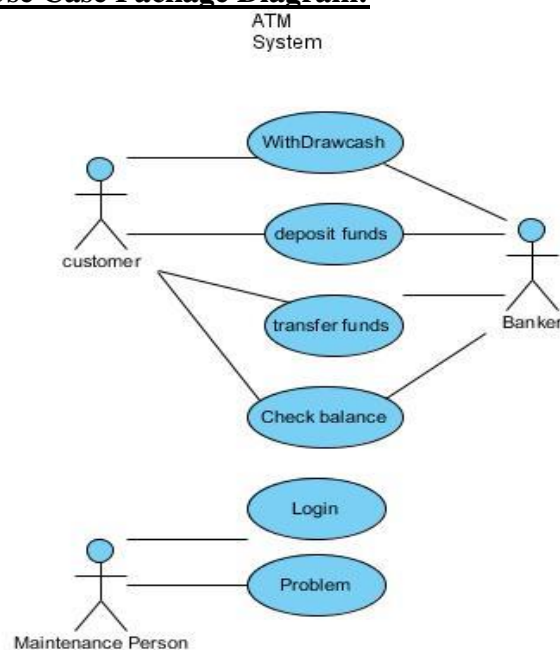
## Rules for Merging Packages:

- **Private elements** within the package **do not merge** with the receiving package.
- UML **allows multiple inheritance** in package merge.
- Any **sub packages within the package are added** to the receiving package.
- UML packages Constructs and Primitive Types are **merged** by UML Kernel package.
- Kernal:merging package

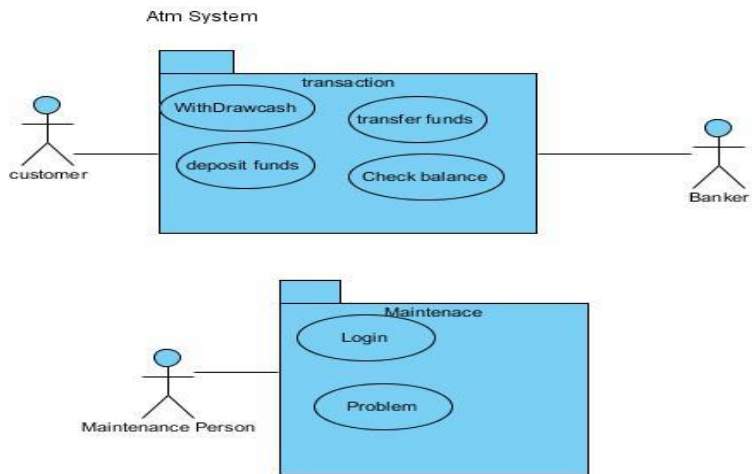
## Package Relationship:

- A relationship between two packages is called a package dependency.
- Dependencies are not transitive.
- The dependency relationship between packages is consistent with the associative relationship between classes.
- Ex. If changing the contents of a package, P2, affects the contents of another package, P1, we can say that P1 has a Package Dependency on P2.

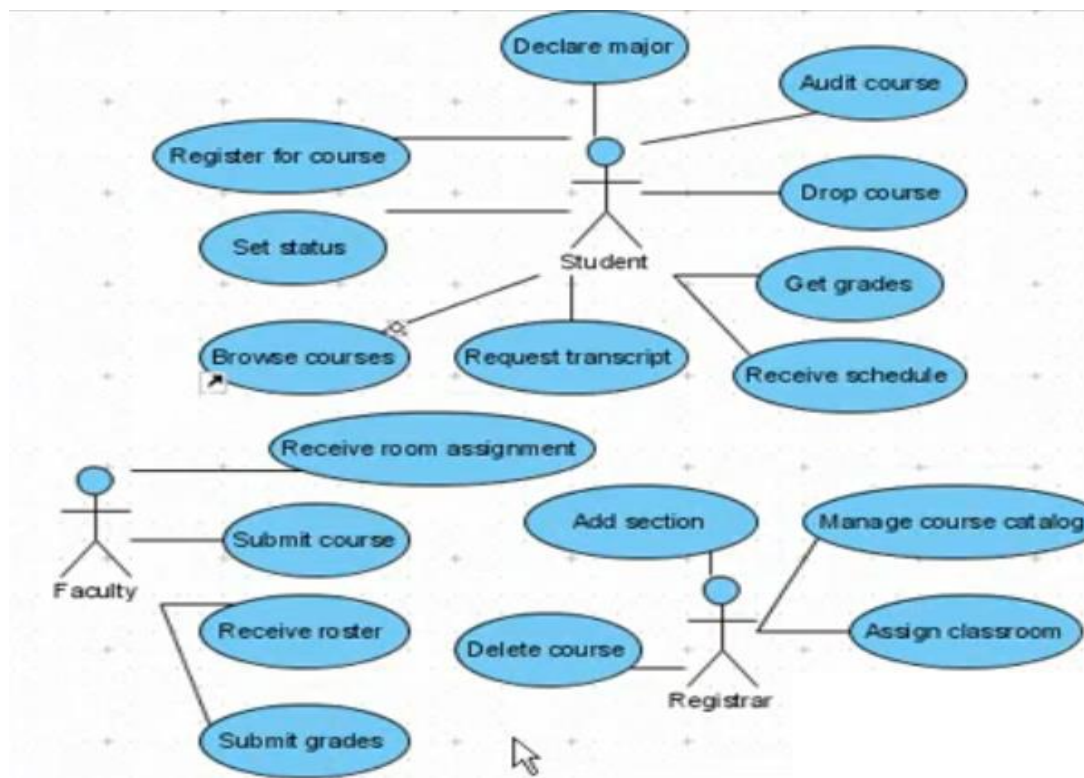
## Use Case Package Diagram:



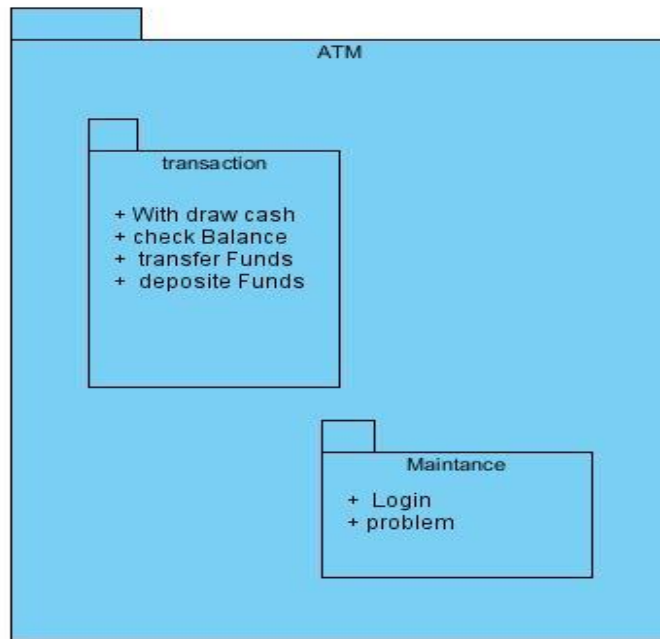
Use case Diagram →→



Use case Package Diagram



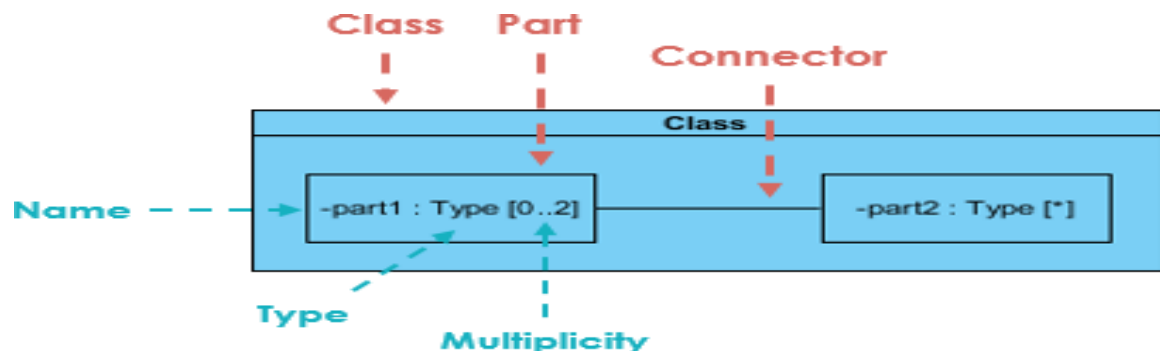
Use Case Package Diagram



**Class Package Diagram**

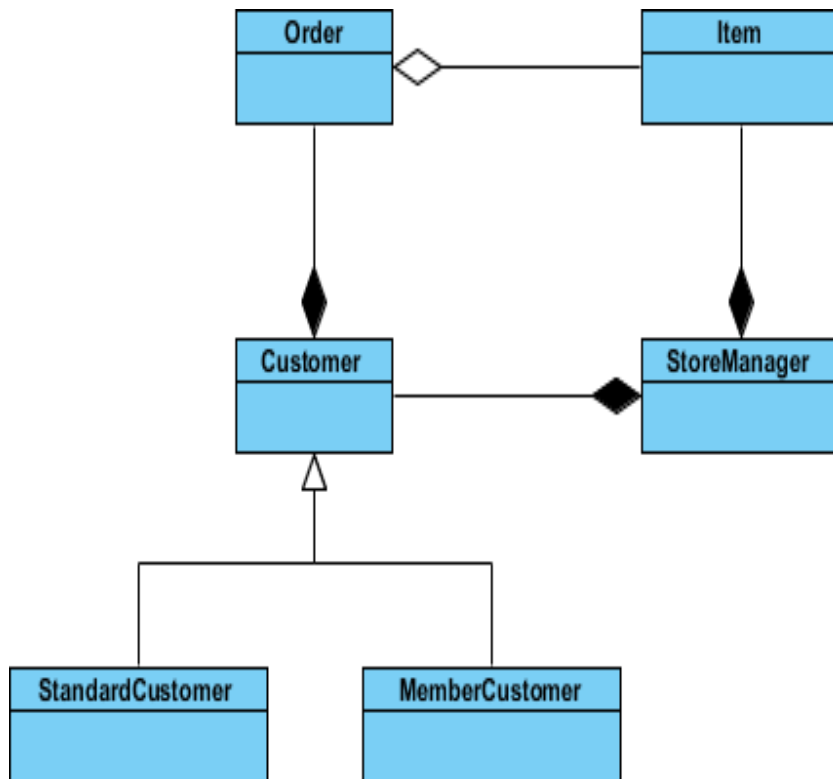
## **2. Composite Structure Diagram**

- Composite Structure Diagram is added to UML 2.0.
- A composite structure diagram is a UML structural diagram that contains classes, interfaces, packages, and their relationships.
- It shows the internal structure of class diagram.
- Composite Structure Diagrams show the internal parts of a class.
- Parts are named: partName:partType[multiplicity]

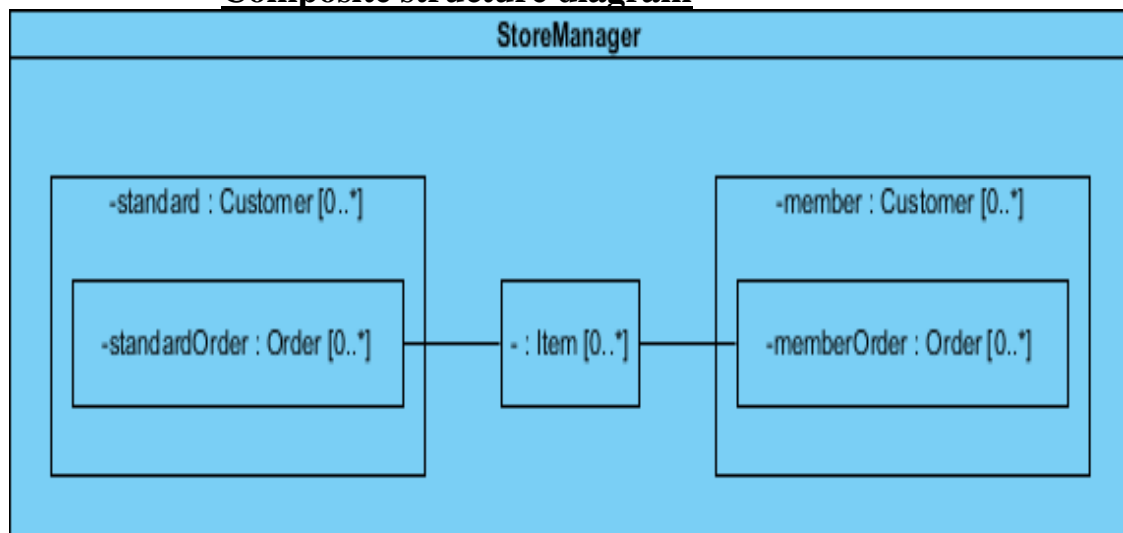


### **Deriving Composite Structure Diagram from Class Diagram:**

- Let's modeling the online store using a class diagram
- We have a class for Item which may be aggregated by the Order class, which is composed by the Customer class which itself is composed by the Store Manager class



### Composite structure diagram



### 3.Component diagram:

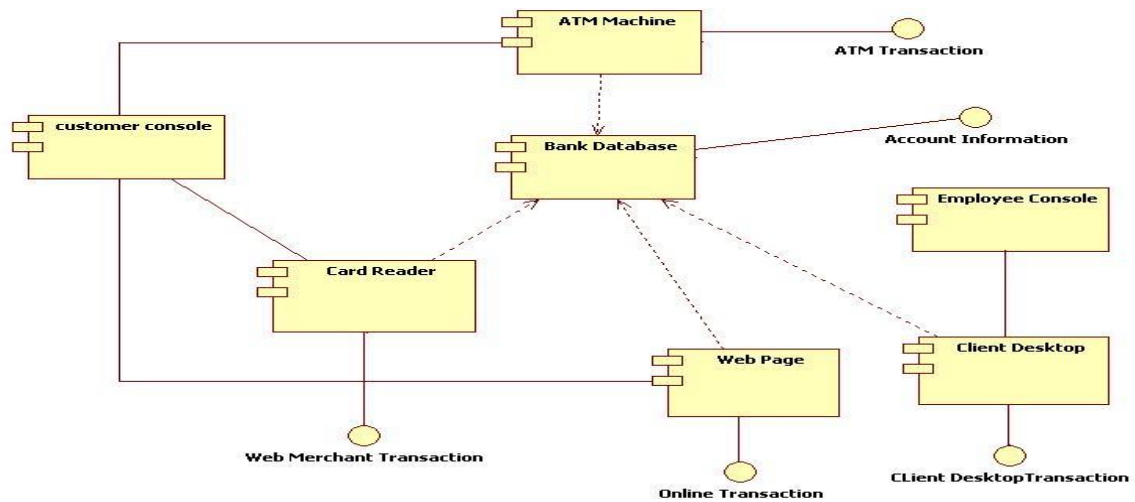
- It shows Structural relationships between the components of a system.
- It model the static implementation view of a system.
- Component diagram is collection of vertices and arcs.
- Used for constructing executable systems through forward and reverse engineering.
- Commonly contain Components, Interfaces and relationships.

Common uses:

1. To model source code



2. To model executable releases
3. To model physical database schema
4. To model adaptable systems



**Ex: Component diagram for ATM**

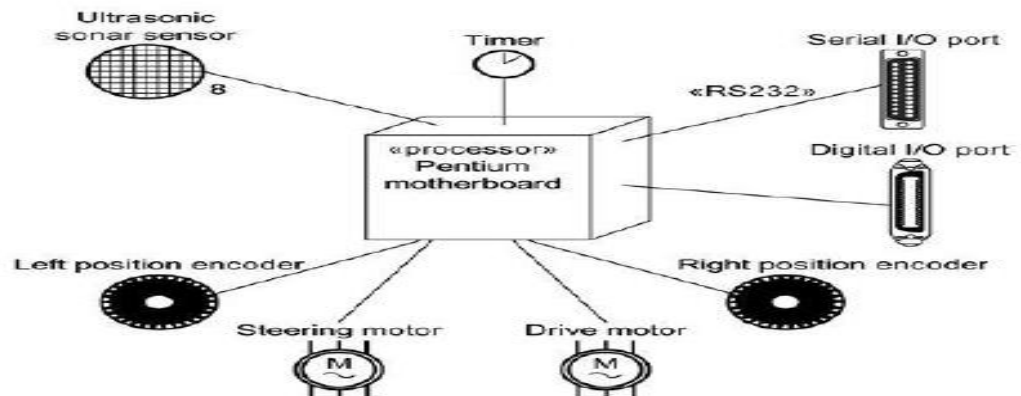
## **4. Deployment diagram:**

- A deployment diagram is a diagram that shows the configuration of run time processing nodes and the components that live on them.
- Graphically, a deployment diagram is a collection of vertices and arcs.
- Deployment diagrams commonly contain Nodes and association relationships. It may also contain notes and constraints.

### **Common modeling Techniques**

To model an embedded system,

- Identify the devices and nodes that are unique to your system.
- Provide visual cues, especially for unusual devices.
- Model the relationships among these processors and devices



**Figure : Modeling an Embedded System**

To model a client/server system,

- Identify the nodes that represent your system's client and server processors.
- Highlight those devices that are independent to the behavior of your system.
- Provide visual cues for these devices via stereotyping.
- Model the topology of these nodes in a deployment diagram.

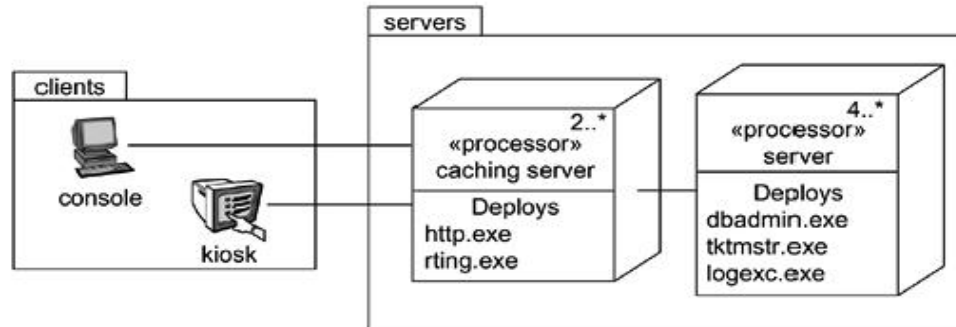


Figure : Modeling a Client/ Server System

### Modeling a Fully Distributed System

- To model a fully distributed system,
- Identify the system's devices and processors as for simpler client/server systems.
- Pay close attention to logical groupings of nodes, which you can specify by using packages.
- Model these devices and processors using deployment diagrams. Where possible, use tools that discover the topology of your system by walking your system's network.
- When modeling a fully distributed system, it's common to reify the network itself as an node. eg:- Internet, LAN, WAN as nodes

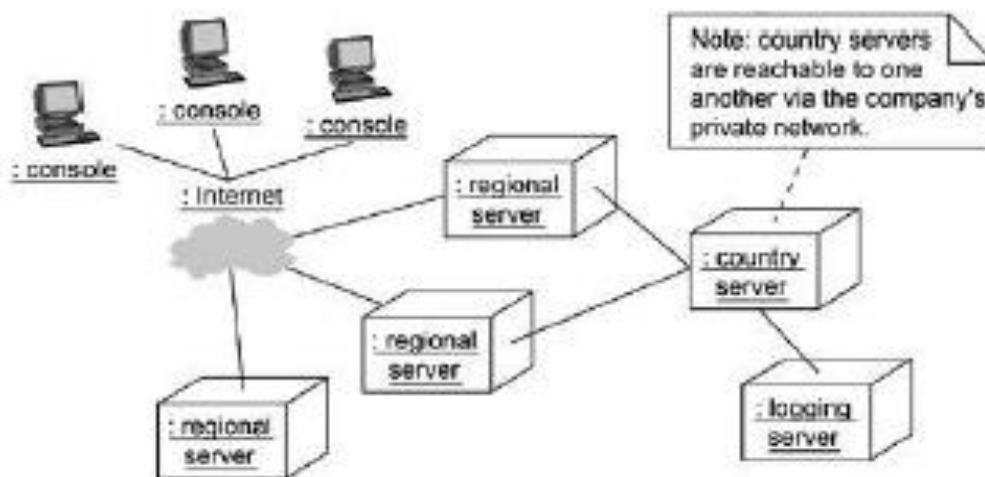


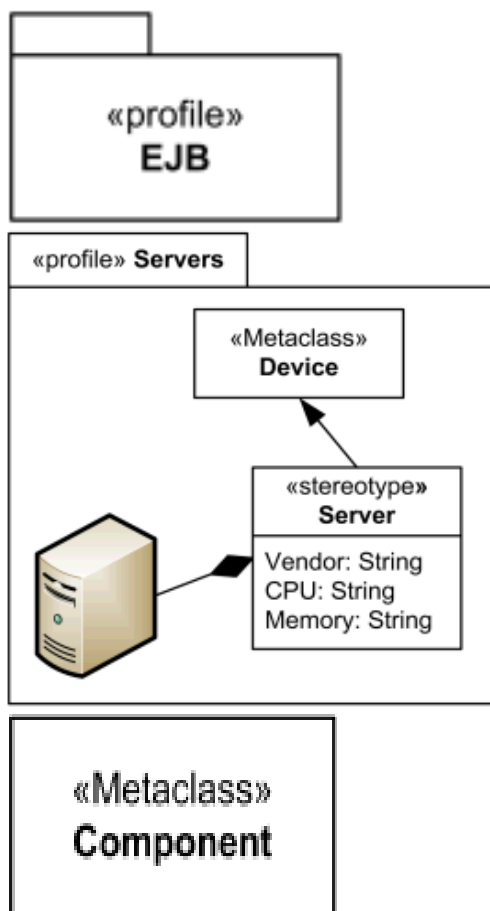
Figure : Modeling a Fully Distributed System

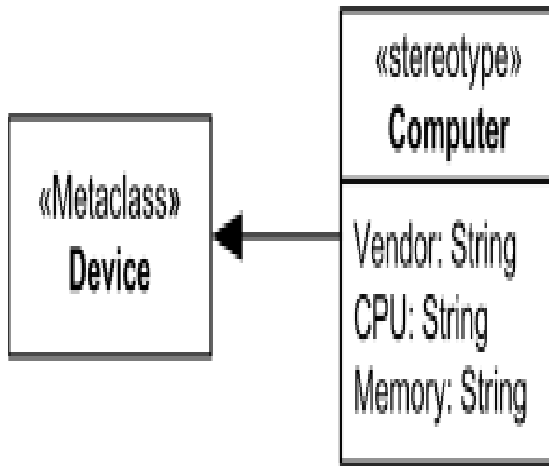
## 5.Profile diagram:

- Profile diagram is [structure diagram](#) which describes lightweight extension mechanism to the UML by defining custom [stereotypes](#), [tagged values](#), and constraints.

### Elements:

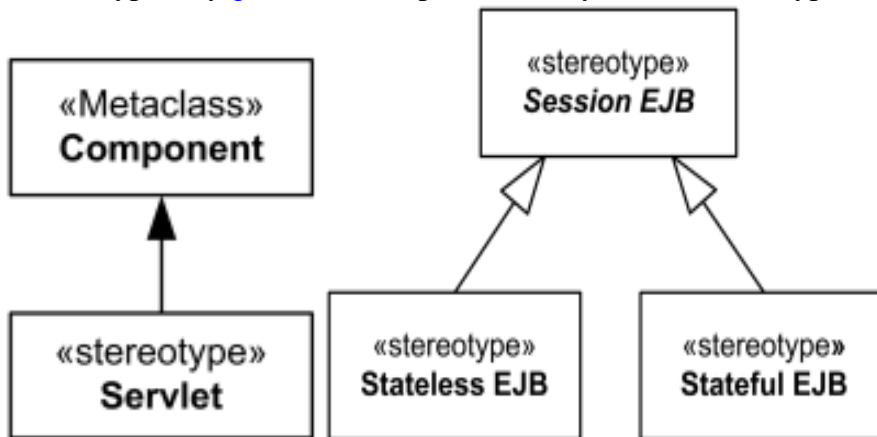
- Profile uses the same notation as a [package](#), with the addition that the keyword «profile» is shown before or above the name of the package.
- A profile can define classes, [stereotypes](#), [data types](#), [primitive types](#), [enumerations](#).
- Metaclass is a profile class and a [packageable element](#) which may be [extended](#) through one or more [stereotypes](#).
- Metaclass may be extended by one or more [stereotypes](#) using special kind of [association - extension](#).
- 





### Stereotype:

- A stereotype uses the same notation as a class, with the addition that the keyword «stereotype» is shown before or above the name of the stereotype.
- A stereotype may [generalize](#) or specialize only another stereotype.



### Previous Questions

1. What is composite diagram? What is its role in modeling? List and explain the components present in it? Explain the same by using Fibonacci sequence
2. Give brief description about the profile diagrams .
3. Give brief description about the Profile diagram.

### Two-marks Questions

1. What is a package? What are the owned elements present in it?
2. What is Composite Structure Diagram
3. What is Component Structure Diagram
4. What is Deployment Diagram

## UNIT- 5

### BEHAVIORAL & ARCHITECTURAL MODELING

Syllabus: Use case Diagrams, Activity Diagrams, State machine Diagrams, Sequence Diagram, Timing Diagram , interaction Over View.

## → Use Case Diagrams

### Terms and Concepts

A *use case diagram* is a diagram that shows a set of use cases and actors and their relationships.

### Common Properties:

A use case diagram is just a special kind of diagram and shares the same common properties as do all other diagrams - a name and graphical contents that are a projection into a model. What distinguishes a use case diagram from all other kinds of diagrams is its particular content.

### Contents:

Use case diagrams commonly contain

- ✓ Use cases
- ✓ Actors
- ✓ Dependency, generalization, and association relationships

Like all other diagrams, use case diagrams may contain *notes* and *constraints*. Use case diagrams may also contain *packages*, which are used to group elements of your model into larger chunks.

### Common Uses:

We apply use case diagrams to model the *static use case view* of a system. When we model the static use case view of a system, we'll typically apply use case diagrams in one of two ways.

1. To model the context of a system

Modeling the context of a system involves drawing a line around the whole system and asserting which actors lie outside the system and interact with it. Here, you'll apply use case diagrams to specify the *actors* and the meaning of their *roles*.

2. To model the requirements of a system

Modeling the requirements of a system involves specifying what that system should do (from a point of view of outside the system), independent of how that system should do it.

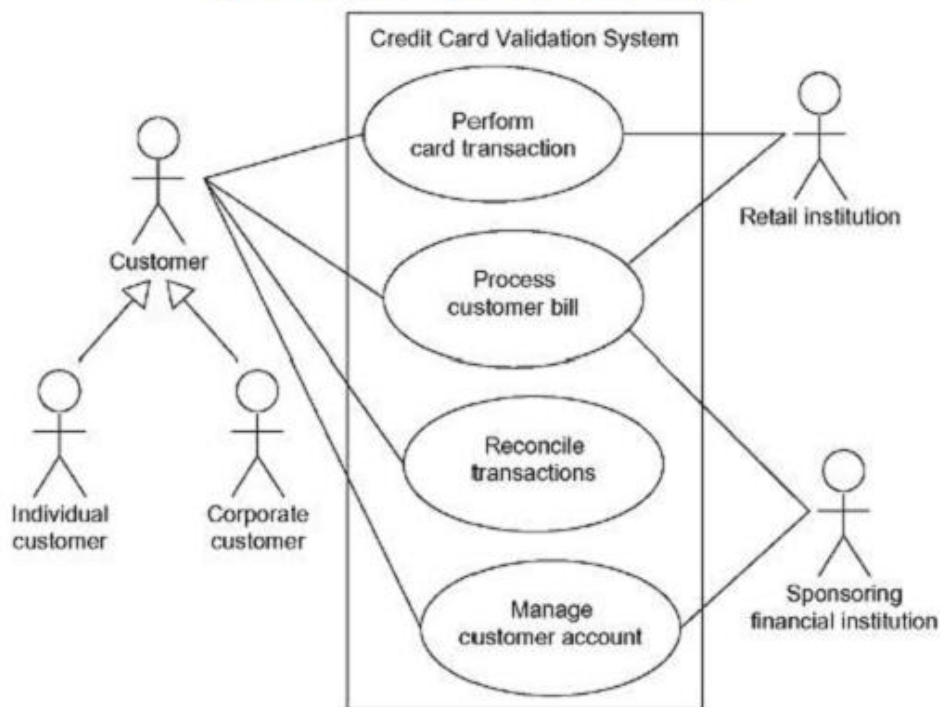
## Common Modeling Techniques of Use case Diagrams:

### Modeling the Context of a System

To model the context of a system,

- Identify the actors that surround the system by considering which groups require help from the system to perform their tasks; which groups are needed to execute the system's functions; which groups interact with external hardware or other software systems; and which groups perform secondary functions for administration and maintenance.
- Organize actors that are similar to one another in a generalization/specialization hierarchy.
- Where it aids understandability, provide a stereotype for each such actor.
- Populate a use case diagram with these actors and specify the paths of communication from each actor to the system's use cases.

Figure 17-2 Modeling the Context of a System



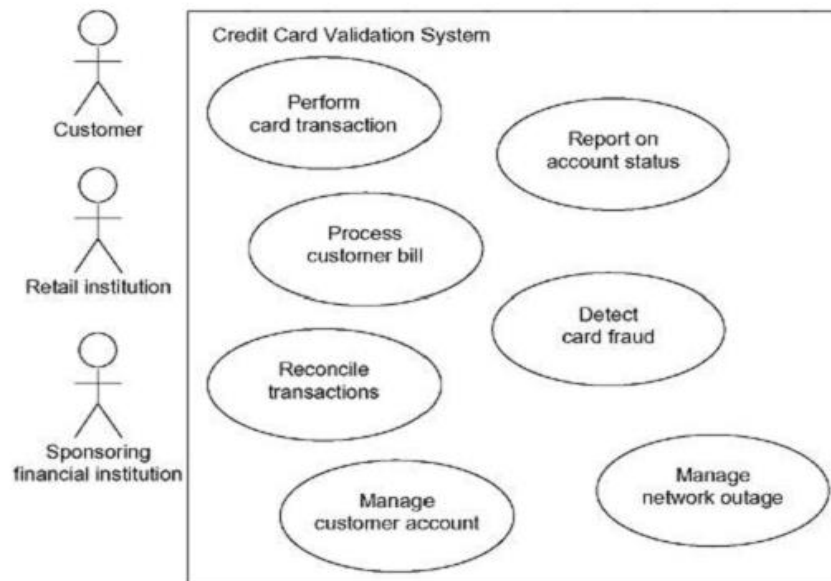
For example, Figure shows the context of a credit card validation system, with an emphasis on the actors that surround the system. we will find customers, of which there are two kinds like Individual customer and Corporate customer. These actors are the roles that human play when interacting with the system. In this context, there are also actors that present other institutions such as retail institution and sponsoring financial institution.

### 2. Modeling the Requirements of a System

A requirement is a design feature, property, or behavior of a system. When you state a system's requirements, you are asserting a contract, established between those things that lie outside the system and the system itself, which declares what you expect that system to do.

To model the requirements of a system,

- Establish the context of the system by identifying the actors that surround it.
- For each actor, consider the behavior that each expects or requires the system to provide.
- Name these common behaviors as use cases.
- Factor common behavior into new use cases that are used by others; factor variant behavior into new use cases that extend more main line flows.
- Model these use cases, actors, and their relationships in a use case diagram.
- Adorn these use cases with notes that assert nonfunctional requirements; you may have to attach some of these to the whole system.



**fig: Modeling the requirements of the system**

The figure shows Detect card fraud is a behaviour important to both the Retail institution and the sponsoring financial institution. Similarly, Report on account status is another behaviour required of the system by the various institutions in its context.

## Forward and Reverse Engineering

Use cases describe how an element behaves, not how that behavior is implemented, so it cannot be directly forward or reverse engineered.

**Forward engineering** is the process of transforming a *model* into *code* through a mapping to an implementation language. A use case diagram can be forward engineered to form tests for the element to which it applies.

To forward engineer a use case diagram,

- For each use case in the diagram, identify its flow of events and its exceptional flow of events.
- Depending on how deeply you choose to test, generate a test script for each flow, using the flow's preconditions as the test's initial state and its postconditions as its success criteria.
- As necessary, generate test scaffolding to represent each actor that interacts with the use case. Actors that push information to the element or are acted on by the element may either be simulated or substituted by its real-world equivalent.
- Use tools to run these tests each time you release the element to which the use case diagram applies.

**Reverse engineering** is the process of transforming *code* into a *model* through a mapping from a specific implementation language. There is a loss of information when moving from a specification of how an element behaves to how it is implemented.

To reverse engineer a use case diagram,

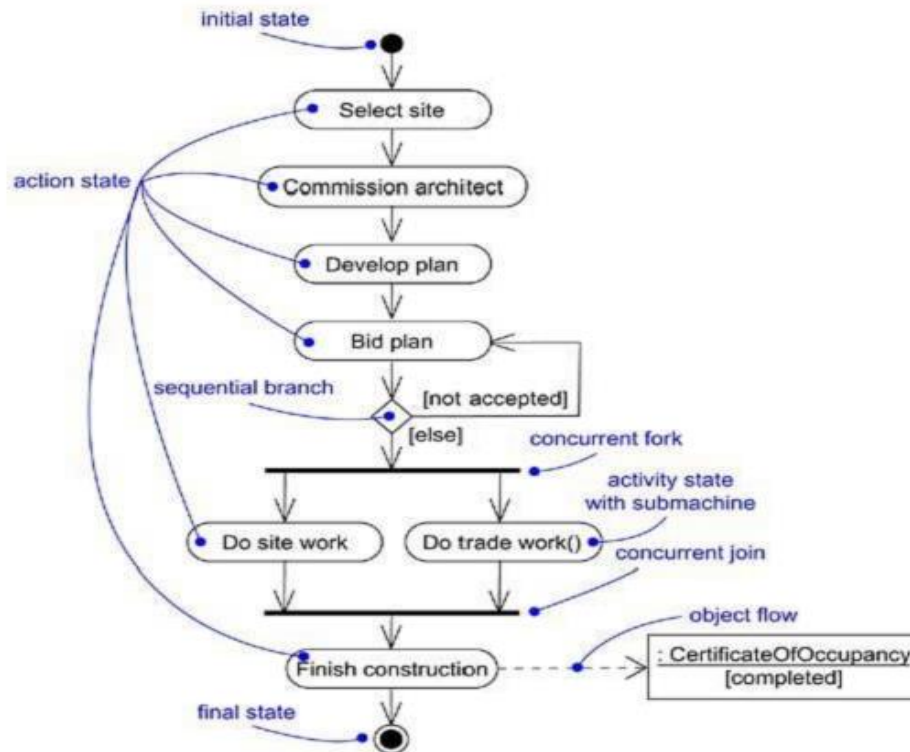
- Identify each actor that interacts with the system.
- For each actor, consider the manner in which that actor interacts with the system, changes the state of the system or its environment, or responds to some event.
- Trace the flow of events in the executable system relative to each actor. Start with primary flows and only later consider alternative paths.
- Cluster related flows by declaring a corresponding use case. Consider modeling variants using extend relationships, and consider modeling common flows by applying include relationships.
- Render these actors and use cases in a use case diagram, and establish their relationships

## → Activity Diagrams

An activity diagram is essentially a **flowchart**, showing flow of control from activity to activity. We use activity diagrams to model the dynamic aspects of a system.

**Figure 19-1 Activity Diagrams**





## Terms and Concepts

An *activity diagram* shows the *flow from activity to activity*. An activity is an *ongoing nonatomic* execution within a *state machine*. Activities ultimately result in some action. Actions encompass calling another operation, sending a signal, creating or destroying an object, or some pure computation, such as evaluating an expression. Graphically, an activity diagram is a collection of *vertices and arcs*.

## Common Properties

An *activity diagram* is just a special kind of diagram and shares the same common properties as do all other diagrams - a name and graphical contents that are a projection into a model. What distinguishes an activity diagram from all other kinds of diagrams is its *content*.

## Contents

Activity diagrams commonly contain

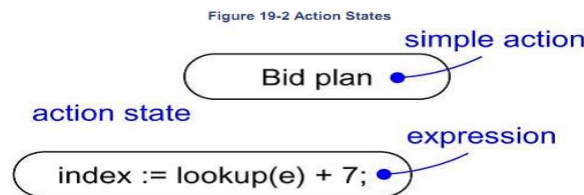
- ✓ Activity states and action states
- ✓ Transitions
- ✓ Objects

Like all other diagrams, activity diagrams may contain notes and constraints.

## Action States and Activity States

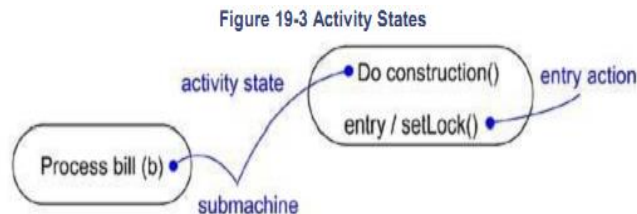
The state represents an action being executed, action states are termed as atomic, executable state i.e., they can't be decomposed. The *executable, atomic computations* are called **action states** because they are states of the system, each representing the execution of an action. As Figure 19-2 shows, you represent an action state using a lozenge shape (a symbol with horizontal top and bottom and convex sides).

Inside that shape, you may write any expression.



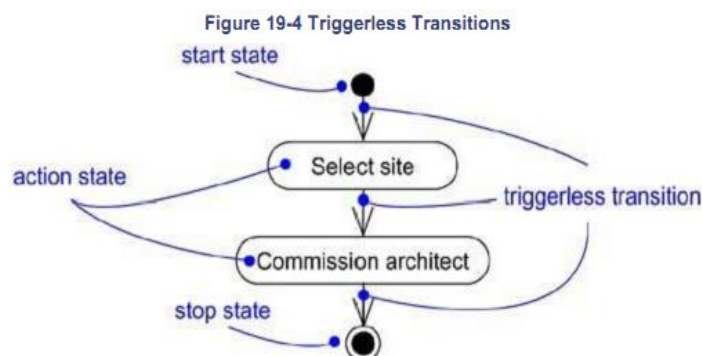
Action states **can't be decomposed**. Furthermore, action states are *atomic*, meaning that events may occur, but the work of the action state is not interrupted.

In contrast, **activity states can be further decomposed**, their activity being represented by other activity diagrams. Furthermore, activity states are *not atomic*, meaning that they may be interrupted and, in general, are considered to take some duration to complete. An activity state as a **composite**, whose flow of control is made up of other *activity states* and *action states*. As Figure shows, there's no notational distinction between action and activity states, except that an activity state may have additional parts, such as *entry* and *exit* actions and *submachine* specifications.



## Transitions

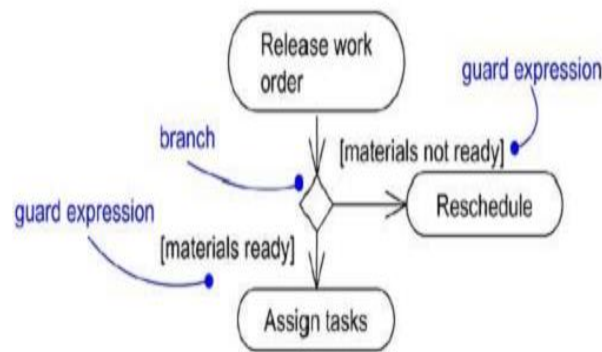
When the *action* or *activity* of a state completes, flow of control passes immediately to the next action or activity state. We specify this flow by using transitions to show the path from *one action or activity state to the next action or activity state*. In the UML, we represent a transition as a simple directed line, as Figure shows.



Indeed, a flow of control has to *start* and *end* someplace. Therefore, as the figure shows, you may specify this *initial state* (*a solid ball*) and *stop state* (*a solid ball inside a circle*).

## Branching

Branch defines decision taken regarding different paths depending on Boolean expression. As in a flowchart, you can include a *branch*, which specifies alternate paths taken based on some Boolean expression. As Figure shows, you represent a branch as a ***diamond***. A branch may have one incoming transition and two or more outgoing ones. On each outgoing transition, you place a Boolean expression, which is evaluated only once on entering the branch. Across all these outgoing transitions, guards should not overlap, but they should cover all possibilities.

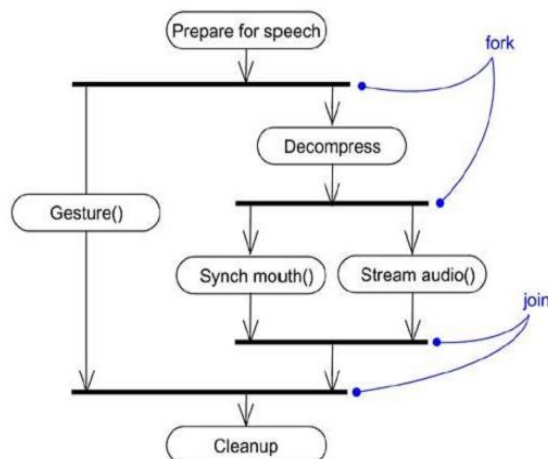


## Forking and Joining

Forking is a process of *splitting* a single flow of control into multiple flows of control. In the UML, we use a ***synchronization bar*** to specify the *forking* and *joining* of these parallel flows of control. A synchronization bar is rendered as a *thick horizontal or vertical line*.

A fork may have one incoming transition and two or more outgoing transitions, each of which represents an independent flow of control. Below the fork, the activities associated with each of these paths continue in parallel.

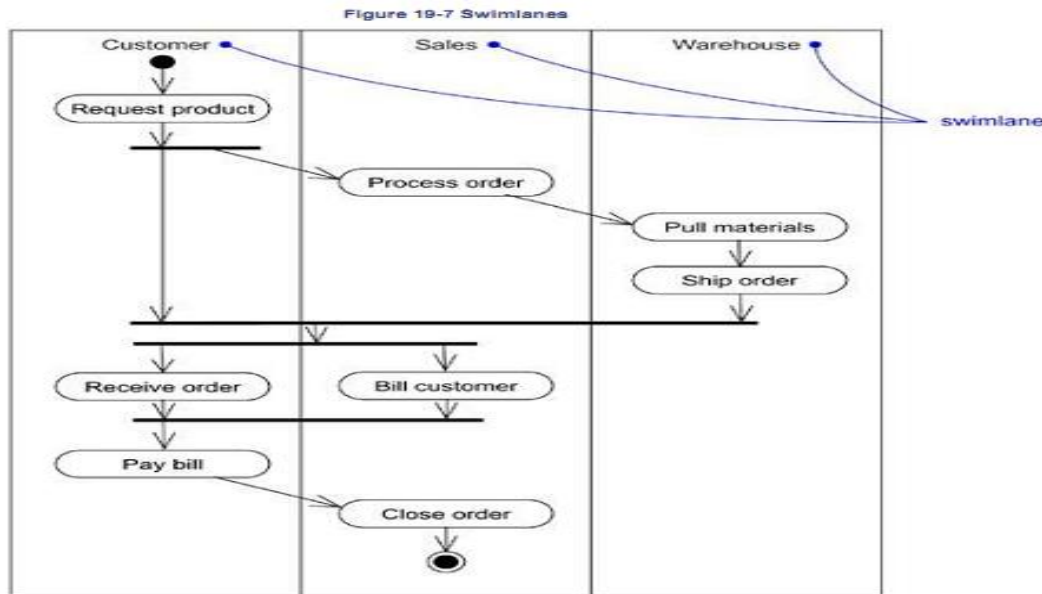
Figure 19-6 Forking and Joining



A **join** represents the *synchronization of two or more concurrent flows of control*. A join may have *two or more incoming transitions* and *one outgoing transition*. Above the join, the activities associated with each of these paths continue in parallel. At the join, the concurrent flows synchronize, meaning that each waits until all incoming flows have reached the join, at which point one flow of control continues on below the join.

## Swimlanes

To partition the activity states on an activity diagram into *groups*, each group representing the responsible for those activities. In the UML, each group is called a **swimlane** because, visually, each group is divided from its neighbor by a vertical solid line, as shown in Figure .



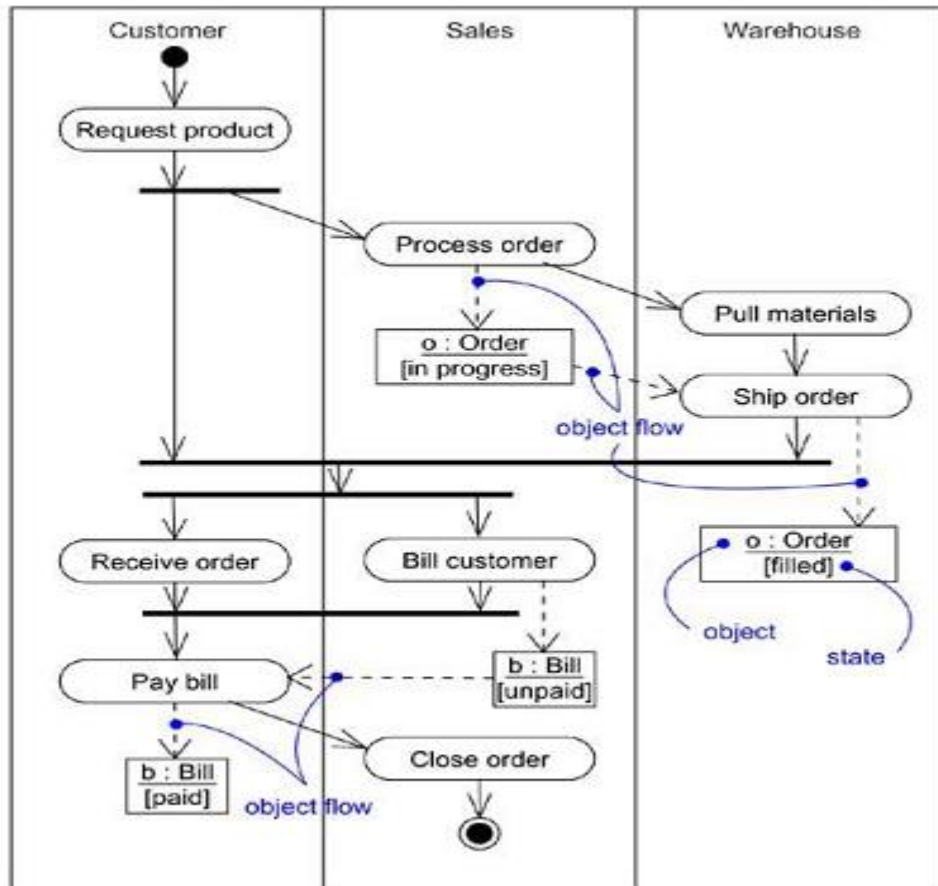
Each swimlane has a name unique within its diagram. A swimlane represent some real-world entity. Each swimlane may eventually be implemented by *one or more classes*. In an activity diagram partitioned into swimlanes, *every activity* belongs to exactly one swimlane, but *transitions* may cross lanes.

## Object Flow

Objects may be involved in the flow of control associated with an activity diagram.

As Figure shows, you can specify the things that are involved in an activity diagram by placing these objects in the diagram, connected using a **dependency** to the activity or transition that creates, destroys, or modifies them. This use of dependency relationships and objects is called an **object flow** because it represents the participation of an object in a flow of control.

**Figure 19-8 Object Flow**



In addition to showing the flow of an object through an activity diagram, you can also show how its role, state and attribute values change. As shown in the figure, you represent the state of an object by naming its state in *brackets* below the object's name. Similarly, you can represent the value of an object's attributes by rendering them in a *compartment* below the object's name.

## Common Uses

We use activity diagrams to model the dynamic aspects of a system. When we use an activity diagram to model some dynamic aspect of a system, we can do so in the context of virtually any modeling element. When we model the dynamic aspects of a system, you'll typically use activity diagrams in two ways.

1. To model a workflow

Here we'll focus on activities as viewed by the actors that collaborate with the system. Workflows often lie on the fringe of software-intensive systems and are used to visualize, specify, construct, and document business processes that involve the system you are developing. In this use of activity diagrams, modeling *object flow* is particularly important.

2. To model an operation

Here we'll use activity diagrams as flowcharts, to model the details of a computation. In this use of activity diagrams, the modeling of branch, fork, and join states is particularly important. The context of an activity diagram used in this way involves the parameters of the *operation* and its local objects.

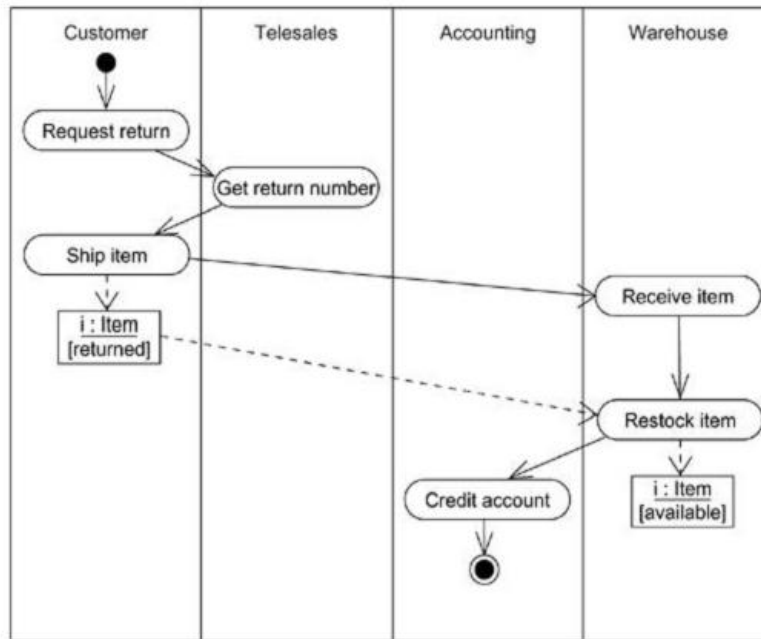
## **Common Modeling Techniques of Activity Diagram:**

### **Modeling a Workflow**

To model a workflow,

- Establish a focus for the workflow. For nontrivial systems, it's impossible to show all interesting workflows in one diagram.
- Select the business objects that have the high-level responsibilities for parts of the overall workflow. These may be real things from the vocabulary of the system, or they may be more abstract. In either case, create a swimlane for each important business object.
- Identify the preconditions of the workflow's initial state and the postconditions of the workflow's final state. This is important in helping you model the boundaries of the workflow.
- Beginning at the workflow's initial state, specify the activities and actions that take place over time and render them in the activity diagram as either activity states or action states.
- For complicated actions, or for sets of actions that appear multiple times, collapse these into activity states, and provide a separate activity diagram that expands on each.
- Render the transitions that connect these activity and action states. Start with the sequential flows in the workflow first, next consider branching, and only then consider forking and joining.
- If there are important objects that are involved in the workflow, render them in the activity diagram, as well. Show their changing values and state as necessary to communicate the intent of the object flow.

**Figure 19-9 Modeling a Workflow**



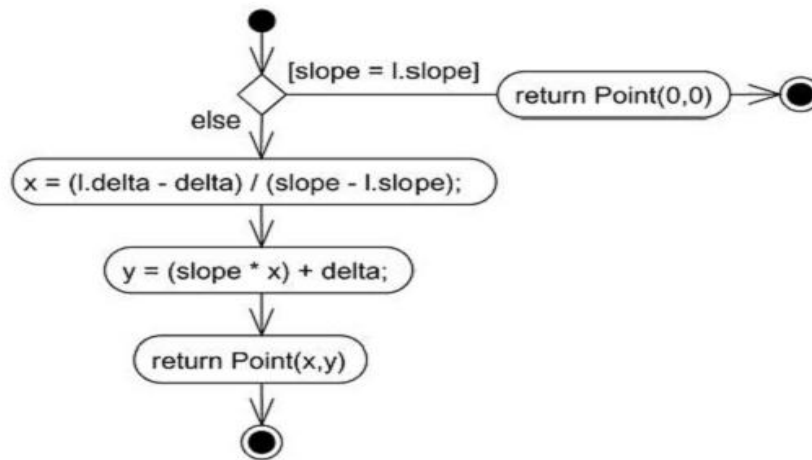
For example, Figure shows an activity diagram for a retail business, which specifies the workflow involved when a customer returns an item from a mail order.

## Modeling an Operation

To model an operation,

- Collect the abstractions that are involved in this operation. This includes the operation's parameters (including its return type, if any), the attributes of the enclosing class, and certain neighboring classes.
- Identify the preconditions at the operation's initial state and the postconditions at the operation's final state. Also identify any invariants of the enclosing class that must hold during the execution of the operation.
- Beginning at the operation's initial state, specify the activities and actions that take place over time and render them in the activity diagram as either activity states or action states.
- Use branching as necessary to specify conditional paths and iteration.
- Only if this operation is owned by an active class, use forking and joining as necessary to specify parallel flows of control.

For example, in the context of the class `Line`, Figure shows an activity diagram that specifies the algorithm of the operation `intersection`, whose signature includes one parameter (`l`, an in parameter of the class `Line`) and one return value (of the class `Point`).



**Figure : Modeling an Operation**

## Forward and Reverse Engineering

**Forward engineering** (the creation of code from a model) is possible for activity diagrams, especially if the context of the diagram is an *operation*. For example, using the previous activity diagram, a forward engineering tool could generate the following C++ code for the operation intersection.

```

Point Line::intersection (l : Line) {
    if (slope == l.slope) return Point(0,0);

    int x = (l.delta - delta) / (slope - l.slope);

    int y = (slope * x) + delta;

    return Point(x, y);
}
  
```

**Reverse engineering** (the creation of a model from code) is also possible for activity diagrams, especially if the context of the code is the *body of an operation*. In particular, the previous diagram could have been generated from the implementation of the class Line.

More interesting than the reverse engineering of a model from code is the animation of a model against the execution of a deployed system.

## → State Machine Diagrams

A *statechart diagram* shows a state machine, emphasizing the flow of control from state to state. A *state machine* is a behavior that specifies the sequences of states an object goes through during its lifetime in response to events, together with its responses to those events.



A *transition* is a relationship between two states indicating that an object in the first state will perform certain actions and enter the second state when a specified event occurs and specified conditions are satisfied.

- Graphically, a statechart diagram is a collection of *vertices* and *arcs*.

## Common Properties

A *statechart diagram* is just a special kind of diagram and shares the same common properties as do all other diagrams - that is, a name and graphical contents that are a projection into a model. What distinguishes a statechart diagram from all other kinds of diagrams is its *content*.

### Contents :

State chart diagrams commonly contain

- Simple states and composite states
- Transitions, including events and actions

Like all other diagrams, statechart diagrams may contain *notes* and *constraints*.

### Common Uses :

State chart diagrams are used to model the dynamic aspects of a system. When we model the dynamic aspects of a system, a class, or a use case, we'll typically use statechart diagrams in one way.

- **To model reactive objects**

A *reactive* object is typically idle until it receives an event. When it receives an event, its response usually depends on previous events. After the object responds to an event, it becomes idle again, waiting for the next event. For these kinds of objects, we'll focus on the *stable states* of that object, the events that trigger a transition from state to state, and the actions that occur on each state change.

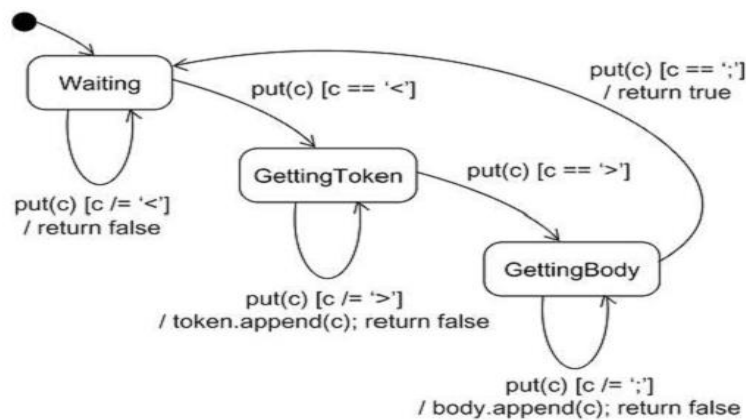
## Common Modeling Techniques of state chart diagram

### 1. Modeling Reactive Objects

To model a reactive object,

- Choose the context for the state machine, whether it is a class, a use case, or the system as a whole.
- Choose the initial and final states for the object. To guide the rest of your model, possibly state the pre- and post conditions of the initial and final states, respectively.
- Decide on the stable states of the object by considering the conditions in which the object may exist for some identifiable period of time. Start with the high-level states of the object and only then consider its possible sub states.
- Decide on the meaningful partial ordering of stable states over the lifetime of the object.
- Decide on the events that may trigger a transition from state to state. Model these events as triggers to transitions that move from one legal ordering of states to another.
- Attach actions to these transitions and/or to these states.
- Consider ways to simplify your machine by using substates, branches, forks, joins, and history states.
- Check that all states are reachable under some combination of events.

- Check that no state is a dead end from which no combination of events will transition the object out of that state.
  - Trace through the state machine, either manually or by using tools, to check it against expected sequences of events and their responses.
- For example, Figure - shows the statechart diagram.



**Figure - Modeling Reactive Objects**

## 2. Forward and Reverse Engineering

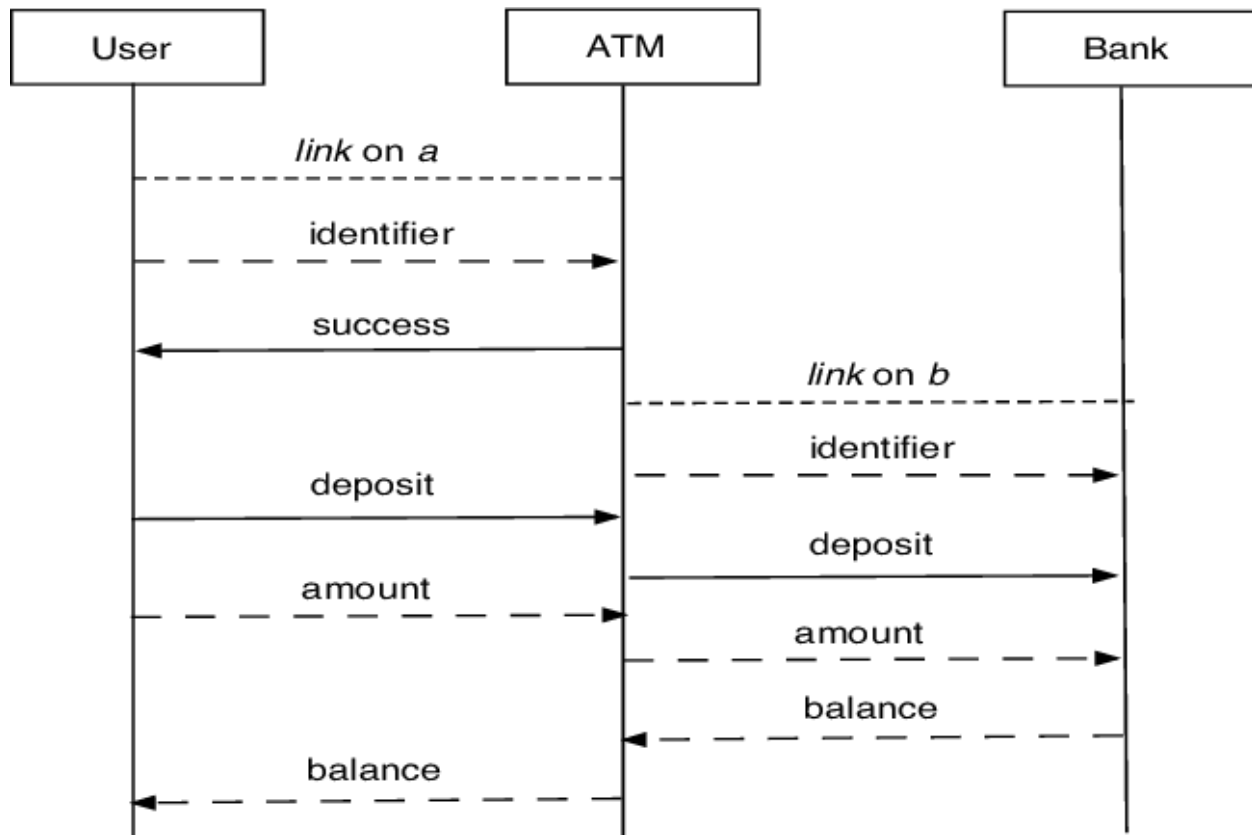
*Forward engineering* (the creation of code from a model) is possible for statechart diagrams, especially if the context of the diagram is a class. The forward engineering tool must generate the necessary private attributes and final static constants.

*Reverse engineering* (the creation of a model from code) is theoretically possible, but practically not very useful. The choice of what constitutes a meaningful state is in the eye of the designer. Reverse engineering tools have no capacity for abstraction and therefore cannot automatically produce meaningful statechart diagrams. More interesting than the reverse engineering of a model from code is the animation of a model against the execution of a deployed system.

## Sequence Diagram:

A sequence diagram shows object interactions arranged in time sequence. It depicts the objects and classes involved in the scenario and the sequence of messages exchanged between the objects needed to carry out the functionality of the scenario. Sequence diagrams are typically associated with use case realizations in the Logical View of the system under development. Sequence diagrams are sometimes called **event diagrams** or **event scenarios**.

A sequence diagram shows, as parallel vertical lines (*lifelines*), different processes or objects that live simultaneously, and, as horizontal arrows, the messages exchanged between them, in the order in which they occur. This allows the specification of simple runtime scenarios in a graphical manner.



#### Diagram building blocks:

If the lifeline is that of an object, it demonstrates a role. Leaving the instance name blank can represent anonymous and unnamed instances.

Messages, written with horizontal arrows with the message name written above them, display interaction. Solid arrow heads represent synchronous calls, open arrow heads represent asynchronous messages, and dashed lines represent reply messages.<sup>[1]</sup> If a caller sends a synchronous message, it must wait until the message is done, such as invoking a subroutine. If a caller sends an asynchronous message, it can continue processing and doesn't have to wait for a response. Asynchronous calls are present in multithreaded applications, event-driven applications and in message-oriented middleware. Activation boxes, or method-call boxes, are opaque rectangles drawn on top of lifelines to represent that processes are being performed in response to the message.

Objects calling methods on themselves use messages and add new activation boxes on top of any others to indicate a further level of processing. If an object is destroyed (removed from memory), an X is drawn on bottom of the lifeline, and the dashed line ceases to be drawn below it. It should be the result of a message, either from the object itself, or another.

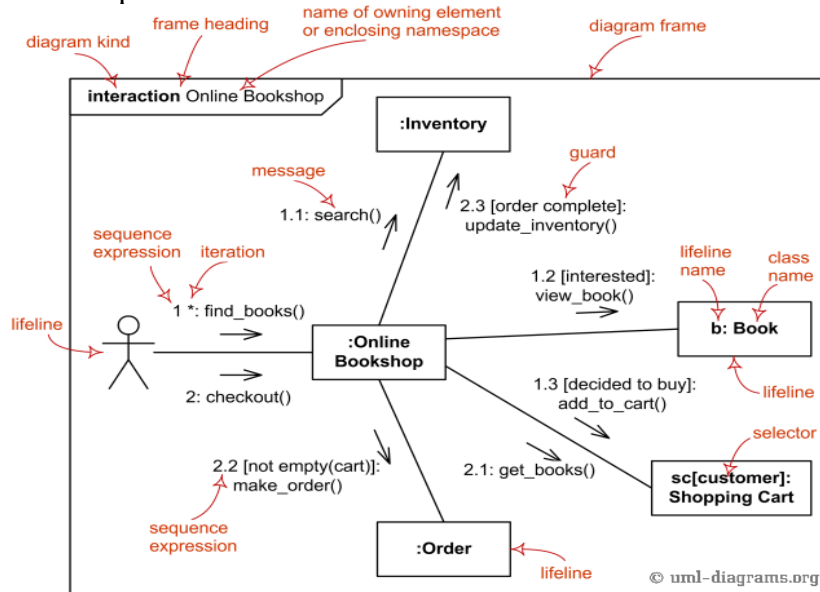
A message sent from outside the diagram can be represented by a message originating from a filled-in circle (*found message* in UML) or from a border of the sequence diagram (*gate* in UML).

UML has introduced significant improvements to the capabilities of sequence diagrams. Most of these improvements are based on the idea of *interaction fragments*<sup>[2]</sup> which represent smaller pieces of an enclosing interaction. Multiple interaction fragments are combined to create a variety of *combined fragments*,<sup>[3]</sup> which are then used to model interactions that include parallelism, conditional branches, optional interactions.

# Communication diagram

Communication diagram (called collaboration diagram in UML 1.x) is a kind of UML interaction diagram which shows interactions between objects and/or parts (represented as lifelines) using sequenced messages in a free-form arrangement.

Communication diagram corresponds (i.e. could be converted to/from or replaced by) to a simple sequence diagram without structuring mechanisms such as interaction uses and combined fragments. It is also assumed that message overtaking (i.e., the order of the receptions are different from the order of sending of a given set of messages) will not take place or is irrelevant. The following nodes and edges are drawn in a UML communication diagrams: frame, lifeline, message. These major elements of the communication diagram are shown on the picture below.

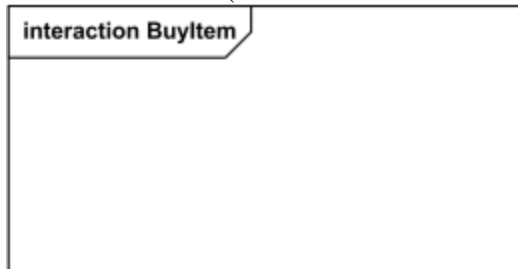


The major elements of UML communication diagram.

## Frame

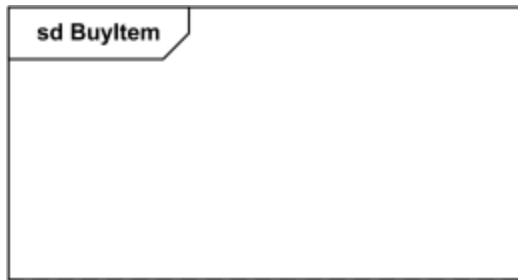
Communication diagrams could be shown within a rectangular frame with the name in a compartment in the upper left corner.

There is no specific long form name for communication diagrams heading types. The long form name interaction (used for interaction diagrams in general) could be used.



Interaction Frame for Communication Diagram BuyItem

There is also no specific short form name for communication diagrams. Short form name sd (which is used for interaction diagrams in general) could be used. This sd is bit confusing as it looks like abbreviation of sequence diagram.



Sd Frame for Communication Diagram BuyItem

## Lifeline

Lifeline is a specialization of named element which represents an individual participant in the interaction. While parts and structural features may have multiplicity greater than 1, lifelines represent only one interacting entity.

If the referenced connectable element is multivalued (i.e, has a multiplicity > 1), then the lifeline may have an expression (selector) that specifies which particular part is represented by this lifeline. If the selector is omitted, this means that an arbitrary representative of the multivalued connectable element is chosen.

A Lifeline is shown as a rectangle (corresponding to the "head" in sequence diagrams). Lifeline in sequence diagrams does have "tail" representing the line of life whereas "lifeline" in communication diagram has no line, just "head".

Information identifying the lifeline is displayed inside the rectangle in the following format:

```
lifeline-ident ::= ([ connectable-element-name [ '[' selector ']' ] ]           [: class-
name ] [decomposition] )                                           | 'self'
```

selector ::= expression

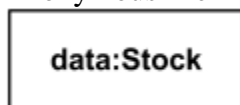
decomposition ::= 'ref' interaction-ident [ 'strict' ]

where class-name is type referenced by the represented connectable element. Note that, although the syntax allows it, lifeline-ident cannot be empty.

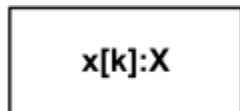
The lifeline head has a shape that is based on the classifier for the part that this lifeline represents. Usually the head is a white rectangle containing name of the class after colon.



Anonymous lifeline of class User.



Lifeline "data" of class Stock



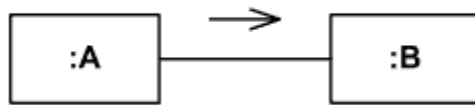
Lifeline "x" of class X is selected with selector [k].

If the name is the keyword self, then the lifeline represents the object of the classifier that encloses the Interaction that owns the Lifeline. Ports of the enclosure may be shown separately even when self is included.

## Message

Message in communication diagram is shown as a line with sequence expression and arrow above the line. The arrow indicates direction of the communication.

#### 1.2.4 [s1.equals(s2)]: remove()



Instance of class A sends remove() message to instance of B if s1 is equal to s2

Sequence Expression

The sequence expression is a dot separated list of sequence terms followed by a colon (":") and message name after that:

sequence-expression ::= sequence-term '.' ... '.' message-name

For

example,

3b.2.2:m5

contains sequence expression 3b.2.2 and message name m5.

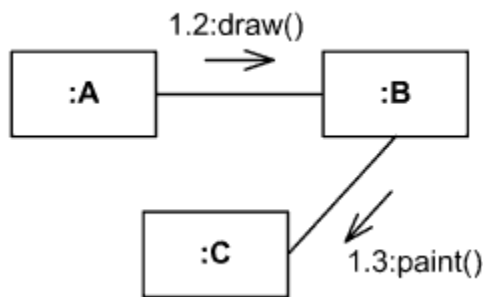
Each sequence term represents a level of procedural nesting within the overall interaction. Each sequence-term has the following syntax:

sequence-term ::= [ integer [ name ] ] [ recurrence ]

The integer represents the sequential order of the message within the next higher level of procedural calling (activation). Messages that differ in one integer term are sequential at that level of nesting.

For example,

- message with sequence 2 follows message with sequence 1,
- 2.1 follows 2
- 5.3 follows 5.2 within activation 5
- 1.2.4 follows message 1.2.3 within activation 1.2.

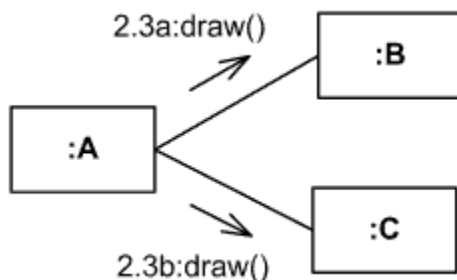


Instance of A sends draw() message to instance of B, and after that B sends paint() to C

The name represents a concurrent thread of control. Messages that differ in the final name are concurrent at that level of nesting.

For example,

- messages 2.3a and 2.3b are concurrent within activation 2.3,
- 1.1 follows 1a and 1b,
- 3a.2.1 and 3b.2.1 follow 3.2.



Instance of A sends draw() messages concurrently to instance of B and to instance of C

The recurrence defines conditional or iterative execution of zero or more messages that are executed depending on the specified condition.

```
recurrence ::= branch | loop
branch ::= '[' guard ']'
loop ::= '*' [ '||' ] [ '['iteration-clause ']' ]
```

A guard specifies condition for the message to be sent (executed) at the given nesting depth. UML does not specify guard syntax, so it could be expressed in pseudocode, some programming language, or something else.

## Timing Diagram

A **timing diagram** in the Unified Modeling Language 2.0 is a specific type of interaction diagram, where the focus is on timing constraints.

Timing diagrams are used to explore the behaviors of objects throughout a given period of time. A timing diagram is a special form of a sequence diagram. The differences between timing diagram and sequence diagram are the axes are reversed so that the time is increased from left to right and the lifelines are shown in separate compartments arranged vertically.

There are two basic flavors of timing diagram: the concise notation, and the robust notation

## Interaction diagrams

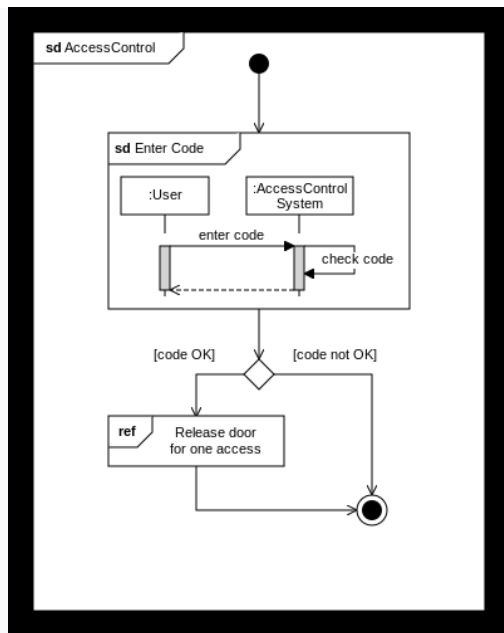
Interaction diagrams, a subset of behavior diagrams, emphasize the flow of control and data among the things in the system being modeled. For example, the [sequence diagram](#) shows how objects communicate with each other regarding a sequence of messages.

## Interaction Diagram Overview

Interaction Overview Diagram is one of the fourteen types of diagrams of the Unified Modeling Language (UML), which can picture a control flow with nodes that can contain interaction diagrams.

The interaction overview diagram is similar to the activity diagram, in that both visualize a sequence of activities. The difference is that, for an interaction overview, each individual activity is pictured as a frame which can contain a nested interaction diagram. This makes the interaction overview diagram useful to "deconstruct a complex scenario that would otherwise require multiple if-then-else paths to be illustrated as a single sequence diagram".

The other notation elements for interaction overview diagrams are the same as for activity diagrams. These include initial, final, decision, merge, fork and join nodes. The two new elements in the interaction overview diagrams are the "interaction occurrences" and "interaction elements"



## **Previous Questions**

1. What is Activity diagram?
2. Give brief description about the State Machine diagrams .
3. Give brief description about the Sequence diagram.
4. Difference Between Communication and Timing Diagram.

## **Two-marks Questions**

1. What is a Usecase? What are the owned elements present in it?
2. What is State machine Diagram
3. What is Interaction Diagram
4. What is Activity Diagram