END TERM EXAMINATION **Roll Number ……..**
SIXTH SEMESTER (B-TECH.)

Paper Code : ETCS 304 Subject : Object Oriented Software Engineering
Time : 3:00 hrs. MM : 75

Note : Attempt any five questions including Q.no. 1, which is compulsory.

Q.1 (a) What is an object? **(2.5*10)**

Ans: An object is anything that exists in the real world, such as a person, a place or a thing. It can be any noun or noun phrase, either physical or conceptual. An object contains data and functions as its integral parts. The data integral to an object are called attributes of the object and the functions are called methods.

(b) Differentiate between class and object.

Ans: A class is a group of similar objects. It is a blueprint that is used to create objects. A single object is simply an instance of a class.

(c) What is Inheritance?

Ans: Inheritance is the process by which objects acquire the properties of objects of other class. When two classes have parent child relationship, the child class (subclass) inherits the properties of the parent class (super class). Inheritance is specified by 'Is-A Relationship'.

(d) What is meant by Encapsulation in OOSD?

Ans: Encapsulation means containing or packaging data within an object. Here data and procedures are tied together to prevent them from external misuse. Encapsulation also means data hiding, which is the process of hiding all aspects of an object that do not contribute to its essential characteristics.

(e) What is meant by Polymrphism in the context of object technology?

Ans: Polymorphism means the ability to take more than one form. It means the same operation behaving differently on different classes. Polymorphism is achieved by method overloading and method overriding.

(f) What are different types of relationships?

Ans: There are three main types of relationships, as given below.

Is-A Relationship: This relationship specifies inheritance.

Has-A Relationship: This relationship is used for Aggregation. Here, a class contains another class as its member.

Uses-A Relationship: This relationship represents Association.

(g) What is Object Identifier (OID)?

Ans: Whenever an object is created, it is uniquely identified in the system by a unique code called OID or sometimes as unique identifier (UID). OID is never changed or deleted even if the objects name or its location is changed or the object is deleted.

(h) Write about the four phases in OMT?

Ans: OMT consists of four phases. These are:

i. Analysis: In this phase an abstraction of what the desired system must do is made in terms of attributes and operations.

ii. System design: The overall architecture and high-level decisions about the system are made.

iii. Object design: It produces a design document, consisting of detailed objects static, dynamic and functional models.

iv. Implementation: This activity produces reusable, extendible, robust code.

(i) What is object model of OMT?

Ans: The object model describes the static structure of the objects in the system and their relationships. It represents the data aspects of the system.

(j) What are different diagrams used in Booch methodology?

Ans: Booch's methodology uses six types of diagrams to depict the underlying design of a system. These are: (1) Class Diagram, (2) Object Diagram, (3) State Transition Diagram, (4) Module Diagram, (5) Process Diagram and (5) Interaction Diagram.

Q.2 (a) Explain various Oops concepts used in object-oriented Software Engineering. (7.5)

Ans . Various concepts of Object Oriented concepts used in OOSE:

• **Object:-** An object is something which is capable of being seen, touched or sensed. Each object has certain distinctions or attributes which enable you to recognize and classify it. Each object has certain actions or methods associated with it.

• **Class:-**A class encapsulates data and procedural abstractions required to describe the content and behavior of some real world entity. A class is a generalized description that describes a collection of similar objects.

• **Encapsulation:-** An object is said to be encapsulate (hide) data and program. This means that the user cannot see the inside of the object but can use the object by calling the program part of the object. This hiding of details of one object from another object, which uses it, is known as encapsulation.

• **Inheritance:-**Inheritance is defined as the property of objects by which instances of a class can have access to data and programs contained in a previously defined class, without those definitions being restated. Classes are linked together in a hierarchy. They form a tree, whose root is the class of "objects". Each class (except for the root class) will have a super class (a class above it in the hierarchy) and possibly subclasses. A class can inherit (i.e. acquire) methods from its super class and in turn can pass methods on to its subclasses. A subclass must have all the properties of the parent class, and other properties as well.

• **Polymorphism:-**Polymorphism includes the ability to use the same message to objects of different classes and have them behave differently. Thus we could define the message "+" for both the addition of numbers and the concatenation (joining) of characters. Polymorphism provides the ability to use the same word to invoke different methods, according to similarity of meaning.

• **Object/class associations:-**Objects/classes interact with each other. Multiplicity defines how many instances of one object/class can be associated with one instance of another object/class.

• **Messages:-**The interaction or communication between the different objects and classes is done by passing messages. The object, which requires communicating with another object, will send a request message to the latter. A message can be sent between two objects only if they have an association between them.

(b) Explain throw-away prototyping and evolutionary prototyping. Discuss the differences between the two. (10)

Ans Throw-Away Prototyping: Also called close ended prototyping. Throwaway or Rapid Prototyping refers to the creation of a model that will eventually be discarded rather than becoming part of the final delivered software. Rapid Prototyping involved creating a working model of various parts of the system at a very early stage, after a relatively short investigation. The method used in building it is usually quite informal, the most important factor being the speed with which the model is provided. The model then becomes the starting point from which users can re-examine their expectations and clarify their requirements. When this has been

achieved, the prototype model is 'thrown away', and the system is formally developed based on the identified requirements. The most obvious reason for using Throwaway Prototyping is that it can be done quickly. If the users can get quick feedback on their requirements, they may be able to refine them early in the development of the software. Speed is crucial in implementing a throwaway prototype, since with a limited budget of time and money little can be expended on a prototype that will be discarded. Strength of throwaway prototyping is its ability to construct interfaces that the users can test. The user interface is what the user sees as the system, and by seeing it in front of them, it is much easier to grasp how the system will work.

Evolutionary prototyping: Evolutionary Prototyping (also known as breadboard prototyping) is quite different from Throwaway Prototyping. The main goal when using Evolutionary Prototyping is to build a very robust prototype in a structured manner and constantly refine it. The reason for this is that the Evolutionary prototype, when built, forms the heart of the new system, and the improvements and further requirements will be built.\ When developing a system using Evolutionary Prototyping, the system is continually refined and rebuilt. "Evolutionary prototyping acknowledges that we do not understand all the requirements and builds only those that are well understood." Evolutionary Prototypes have an advantage over Throwaway Prototypes in that they are functional systems. Although they may not have all the features the users have planned, they may be used on an interim basis until the final system is delivered. In Evolutionary Prototyping, developers can focus themselves to develop parts of the system that they understand instead of working on developing a whole system.

Q.3 (a)How is cyclomatic complexity useful in program test? What is sequence of testing? What is testability? (7.5)

Ans. Cyclomatic complexity measures the amount of decision logic in a single software module. It is used for two related purposes in the structured testing methodology. First, it gives the number of recommended tests for software. Second, it is used during all phases of the software lifecycle, beginning with design, to keep software reliable, testable, and manageable. Cyclomatic complexity is based entirely on the structure of software's control flow graph. The sequence of testing is:

Unit testing: Unit testing is undertaken after a module has been coded and successfully reviewed. Unit testing (or module testing) is the testing of different units (or modules) of a system in isolation.

Integration testing: The primary objective of integration testing is to test the module interfaces, i.e. there are no errors in the parameter passing, when one module invokes another module. During integration testing, different modules of a system are integrated in a planned manner using an integration plan. The integration plan specifies the steps and the order in which modules are combined to realize the full system. After each

integration step, the partially integrated system is tested. System testing System tests are designed to validate a fully developed system to assure that it meets its requirements. There are essentially three main kinds of system testing:

• **Alpha Testing-** Alpha testing refers to the system testing carried out by the test team within the developing organization.

• **Beta testing**-Beta testing is the system testing performed by a select group of friendly customers.

• **Acceptance Testing-**Acceptance testing is the system testing performed by the customer to determine whether he should accept the delivery of the system.

Software testability is the degree to which a software artifact (i.e. a software system, software module, requirements- or design document) supports testing in a given test context. Testability is not an intrinsic property of a software artifact and can not be measure directly. Instead testability is an extrinsic property which results from interdependency of the software to be tested and the test goals, test methods used, and test resources. A lower degree of testability results in increased test effort. In extreme cases a lack of testability may hinder testing parts of the software or software requirements at all

(b) Differentiate between function oriented design and object oriented design. (5)

Ans. Function oriented design:- Function oriented design strategy relies on decomposing the system into a set of interacting functions with a centralized system state shared by these functions. Functions may also maintain local state information but only for the duration of their execution. Function oriented design conceals the details of an algorithm in a function but system state information is not hidden. Object oriented design:-Object oriented design transforms the analysis model created using object-oriented analysis into a design model that serves as a blueprint for software construction. It is a design strategy based on information hiding. Object oriented design is concerned with developing an object-oriented model of a software system to implement the identified requirements. Object oriented design establishes a design blueprint that enables a software engineer to define object oriented architecture in a manner that maximizes reuse, thereby improving development speed and end product quality.

Object oriented design Vs function oriented design-

• Unlike function oriented design methods, in OOD, the basic abstractions are not real world function such as sort, display, track etc. but real world entities such as employee, picture, machine etc.

• In OOD, state information is not represented in a centralized shared memory but is distributed among the objects of the system.

Q.4 What is stepwise refinement? Discuss partitioning & abstraction. (12.5)

Ans. Stepwise Refinement:-Stepwise Refinement is a top-down design strategy originally proposed by Niklaus Wirth. A program is developed by successively refining levels of procedural detail. A hierarchy is developed by decomposing a macroscopic statement of function in a stepwise fashion until programming language statements are reached. Refinement is actually a process of elaboration.

We begin with a statement of function that is defined at a high level of abstraction. That is, the statement describes function or information conceptually but provides no information about the internal workings of the function or the internal structure of the information. Refinementcauses the designer to elaborate on the original statement, providing more and more detail as each successive refinement occurs.

Partitioning:-Problems are often too large and complex to be understood as a whole. For this reason, we tend to partition such problems into parts that can be easily understood and establish interfaces between the parts so that overall function can be accomplished.

Partitioning decomposes a problem into its constituent parts. We establish a hierarchical representation of function or information and then partition the uppermost element by:-

1) Exposing increasing detail by moving vertically in the hierarchy or

2) Functionality decomposing the problem by moving horizontally in the hierarchy.

Abstraction:- Abstraction permits one to concentrate on a problem at some level of generalization without regard to irrelevant low level details; use of abstraction also permits one to work with concepts and terms that are familiar in the problem environment without having to transform them to an unfamiliar structure. • It allows considering the modules at the abstract level without worrying about its details.

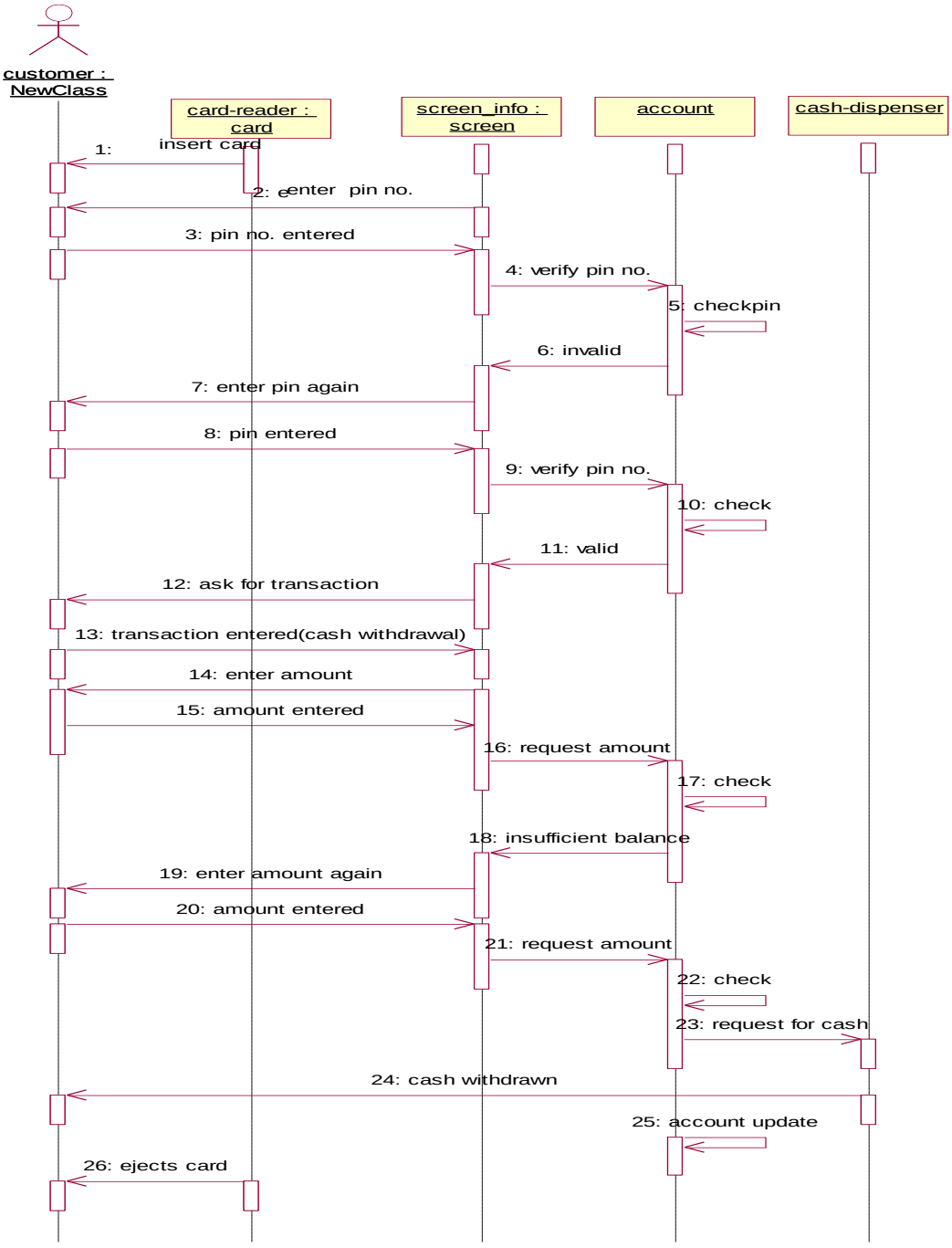
• It provides external behavior of the modules.

• It is used for existing modules as well as for modules that are being design.

• It is essential for the problem partitioning.

Q.5 Draw well labelled S nequence diagram for an ATM System. **(12.5)**

Ans:



SEQUENCE DIAGRAM OF ATM SYSTEM

Q.6 Explain the notation and symbols and relationships used in UCD? Also prepare a UCD for Library Management System. (12.5)

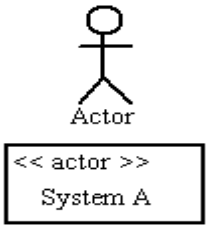
Ans:

Use Case

Draw use cases using ovals. Label with ovals with verbs that represent the system's functions.

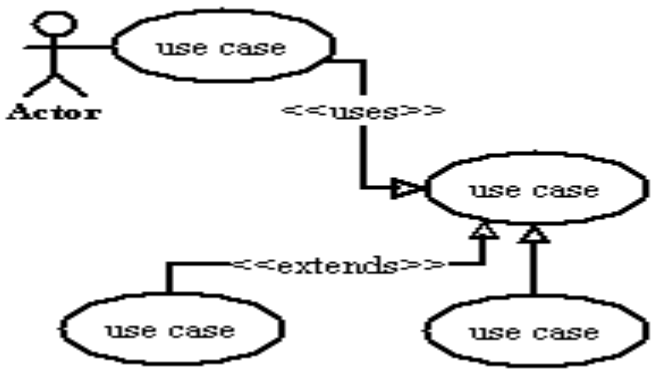
Actors

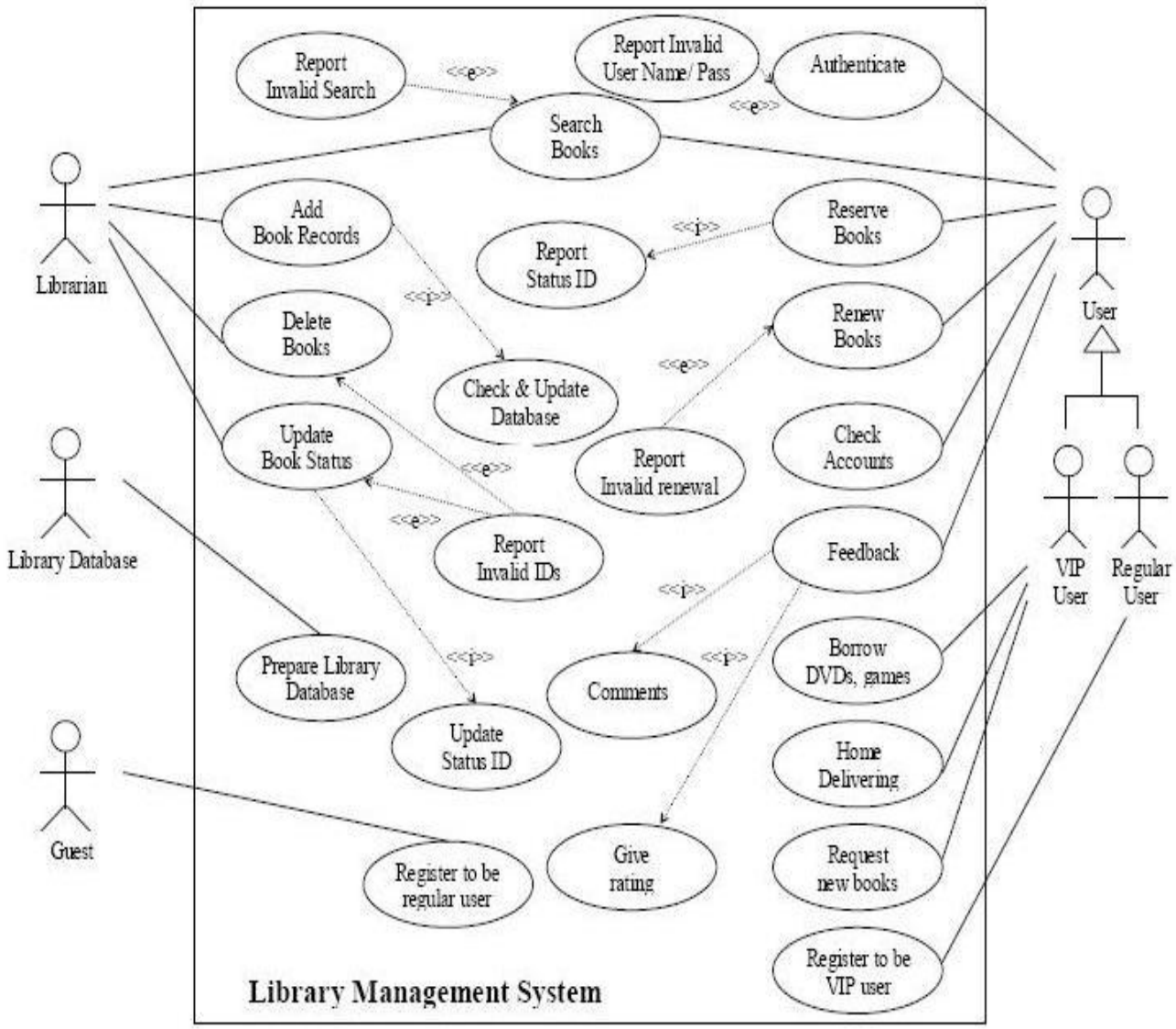
Actors are the users of a system. When one system is the actor of another system, label the actor system with the actor stereotype.



Relationships

Illustrate relationships between an actor and a use case with a simple line. For relationships among use cases, use arrows labeled either "uses" or "extends." A "uses" relationship indicates that one use case is needed by another in order to perform a task. An "extends" relationship indicates alternative options under a certain use case.





Q.7 Explain the following terms w.r.t. the class diagrams in UML. **(12.5)**

(a) Active Class

(b) Visibility

(c) Associations

(d) Multiplicity (Cardinality)

(e) Constraint

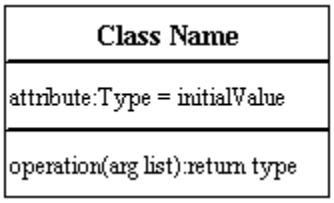
(f) Composition and Aggregation

Ans:

Class Diagrams

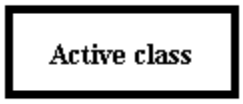
Class diagrams are the backbone of almost every object-oriented method including UML. They describe the static structure of a system. The purpose of a class diagram is to depict the classes within a model. In an object oriented application, classes have attributes (member variables), operations (member functions) and relationships with other classes. The UML class diagram can depict all these things quite easily. The fundamental element of the class diagram is an icon the represents a class. A class icon is simply a rectangle divided into three compartments. The topmost compartment contains the name of the class. The middle compartment contains a list of attributes (member variables), and the bottom compartment contains a list of operations (member functions).

In many diagrams, the bottom two compartments are omitted. Even when they are present, they typically do not show every attribute and operations. The goal is to show only those attributes and operations that are useful for the particular diagram. This ability to abbreviate an icon is one of the hallmarks of UML. Each diagram has a particular purpose. That purpose may be to highlight on particular part of the system, or it may be to illuminate the system in general. The class icons in such diagrams are abbreviated as necessary. There is typically never a need to show every attribute and operation of a class on any diagram.



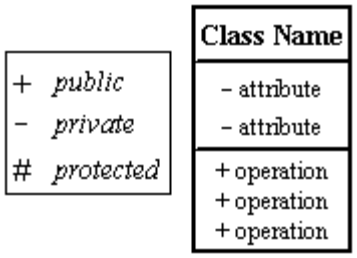
Active Class

Active classes initiate and control the flow of activity, while passive classes store data and serve other classes. Illustrate active classes with a thicker border.



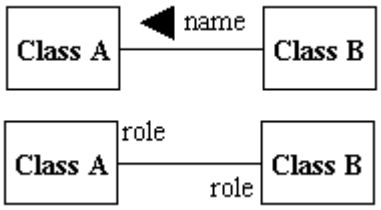
Visibility

Use visibility markers to signify who can access the information contained within a class. Private visibility hides information from anything outside the class partition. Public visibility allows all other classes to view the marked information. Protected visibility allows child classes to access information they inherited from a parent class.



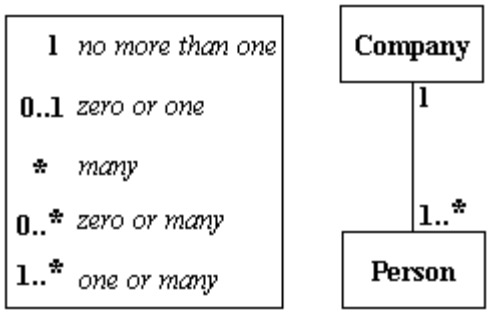
Associations

Associations represent static relationships between classes. Place association names above, on, or below the association line. Use a filled arrow to indicate the direction of the relationship. Place roles near the end of an association. Roles represent the way the two classes see each other. **Note:** It's uncommon to name both the association and the class roles.



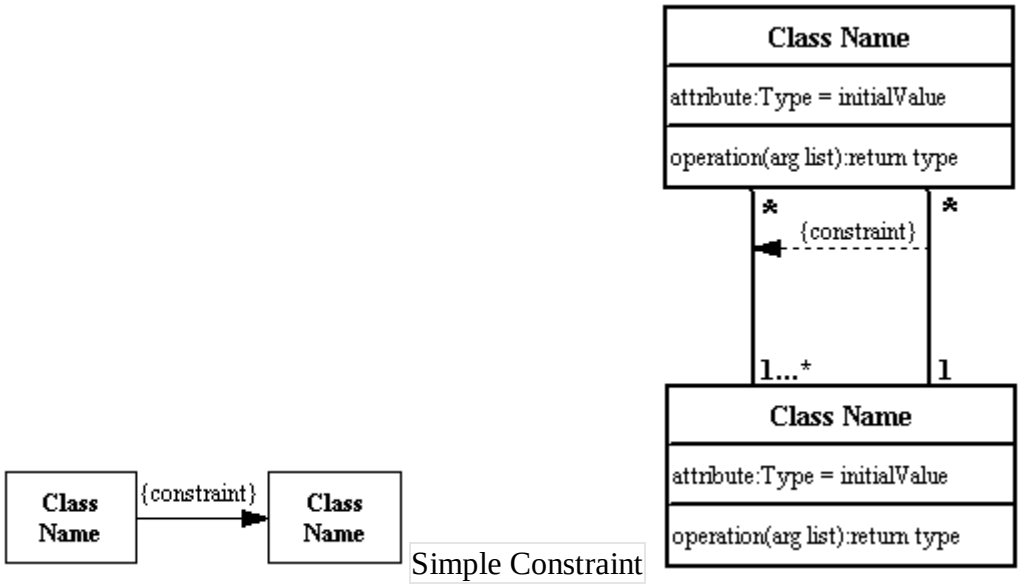
Multiplicity (Cardinality)

Place multiplicity notations near the ends of an association. These symbols indicate the number of instances of one class linked to one instance of the other class. For example, one company will have one or more employees, but each employee works for one company only.



Constraint

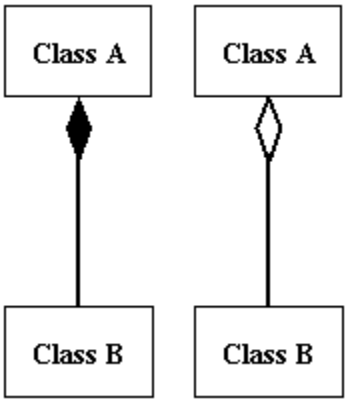
Place constraints inside curly braces {}.



Simple Constraint

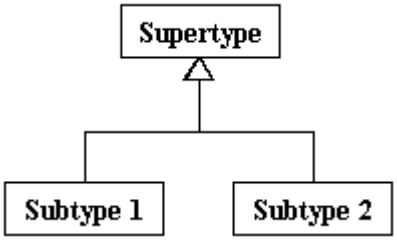
Composition and Aggregation

Composition is a special type of aggregation that denotes a strong ownership between Class A, the whole, and Class B, its part. Illustrate composition with a filled diamond. Use a hollow diamond to represent a simple aggregation relationship, in which the "whole" class plays a more important role than the "part" class, but the two classes are not dependent on each other. The diamond end in both a composition and aggregation relationship points toward the "whole" class or the aggregate.



Generalization

Generalization is another name for inheritance or an "is a" relationship. It refers to a relationship between two classes where one class is a specialized version of another. For example, Honda is a type of car. So the class Honda would have a generalization relationship with the class car.



In real life coding examples, the difference between inheritance and aggregation can be confusing. If you have an aggregation relationship, the aggregate (the whole) can access only the PUBLIC functions of the part class. On the other hand, inheritance allows the inheriting class to access both the PUBLIC and PROTECTED functions of the superclass.

Q.8 Explain UML and its importance in object-oriented systems. (12.5)

Ans:

UML

Unified Modeling Language (**UML**) is a standardized general-purpose modeling language in the field of software engineering. The standard is managed, and was created by, the Object Management Group.

UML includes a set of graphical notation techniques to create visual models of software-intensive systems.

The Unified Modeling Language (UML) is used to specify, visualize, modify, construct and document the artifacts of an object-oriented software intensive system under development. UML offers a standard way to visualize a system's architectural blueprints, including elements such as:

- actors
- business processes
- (logical) components
- activities
- programming language statements
- database schemas, and
- reusable software components.

UML combines best techniques from data modeling (entity relationship diagrams), business modeling (work flows), object modeling, and component modeling. It can be used with all processes, throughout the software development life cycle, and across different implementation technologies. UML has synthesized the notations of the Booch method, the Object-modeling technique (OMT) and Object-oriented software engineering (OOSE) by fusing them into a single, common and widely usable modeling language. UML aims to be a standard modeling language which can model concurrent and distributed systems. UML is a de facto industry standard, and is evolving under the auspices of the Object Management Group (OMG). OMG initially called for information on object-oriented methodologies that might create a rigorous software modeling language. Many industry leaders have responded in earnest to help create the UML standard.[1]

UML models may be automatically transformed to other representations (e.g. Java) by means of QVT-like transformation languages, supported by the OMG. UML is extensible, offering the following mechanisms for

customization: profiles and stereotype. The semantics of extension by profiles have been improved with the UML 2.0 major revision.

The important point to note here is that UML is a 'language' for specifying and not a method or procedure. The UML is used to define a software system; to detail the artifacts in the system, to document and construct - it is the language that the blueprint is written in. The UML may be used in a variety of ways to support a software development methodology (such as the Rational Unified Process) - but in itself it does not specify that methodology or process.

UML defines the notation and semantics for the following domains:

- The User Interaction or Use Case Model - describes the boundary and interaction between the system and users. Corresponds in some respects to a requirements model.

- The Interaction or Communication Model - describes how objects in the system will interact with each other to get work done.

- The State or Dynamic Model - State charts describe the states or conditions that classes assume over time. Activity graphs describe the workflows the system will implement.

- The Logical or Class Model - describes the classes and objects that will make up the system.

- The Physical Component Model - describes the software (and sometimes hardware components) that make up the system.

- The Physical Deployment Model - describes the physical architecture and the deployment of components on that hardware architecture.