UNIT 1 OBJECT MODELING

Structure **Page Nos.**

1.0 INTRODUCTION

In the previous Blocks of this Course we learned the differences between OOA and OOAD and how UML is used for visualizing, specifying, constructing, and documenting.

The goal of object design is to identify the object that the system contains, and the interactions between them. The system implements the specification. The goals of object-oriented design are:

(1) More closely to problem domain

(2) Incremental change easy

(3) Supports reuse. Objects during Object Oriented Analysis OOA focuses on problem or in other word you can say semantic objects. In Object Oriented Design we focuses on defining a solution. Object Oriented modeling is having three phases object modeling, dynamic modeling, and function modeling. In this Unit we will discuss the concepts of object modeling. We will also discuss aggregation, multiple inheritance, generalisation in different form and metadata.

1.1 OBJECTIVES

After going through this unit, you should be able to:

- describe and apply the concept of generalisation;
- understand and apply the concepts abstract Class, multiple Inheritance;
- apply generalisation as an extension;
- apply generalisation as a Restriction, and
- explain the concept of Metadata and constraints.

1.2 ADVANCED MODELING CONCEPTS

You have to follow certain steps for object-oriented design.

These steps for OO Design methodology are:

1) produce the object model
2) produce the dynamic model
3) produce the functional model
4) define the algorithm for major operation
5) optimize and package.

The first step for object-oriented designing is object modeling. Before we go into details of object modeling first of all we should know "what is object modeling"? You can say that **Object** modeling identifies the objects and classes in the problem domain, and identifies the relationship between objects.

In this whole process first of all we have to identify objects, then structures, attributes, associations, and finally services.

1.2.1 Aggregation

Aggregation is a stronger form of association. It represents the **has-a** or **part-of** relationship. An aggregation association depicts a complex object that is composed of other objects. You may characterize a house in terms of its roof, floors, foundation, walls, rooms, windows, and so on. A room may, in turn be, composed of walls, ceiling, floor, windows, and doors, as represented in *Figure 1*.

In UML, a link is placed between the "whole "and "parts" classes, with a diamond head (*Figure1*) attached to the whole class to indicate that this association is an aggregation. Multiplicity can be specified at the end of the association for each of the **part-of** classes to indicate the constituent parts. The process of decomposing a complex object into its component objects can be extended until the desired level of detail is reached.

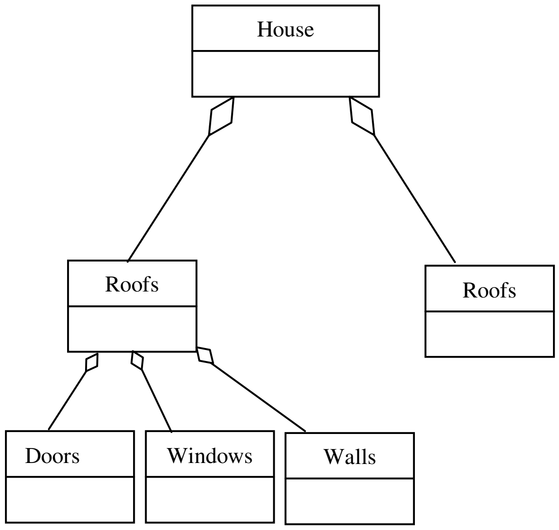

Figure 1: A house and some of its component

You can see that object aggregation helps us describe models of the **real world** that are composed of other models, as well as those that are composed of still other models. Analysts, at the time of describing a complex system of aggregates, need to describe them in enough detail for the system at hand. In the case of a customer order and services, a customer order is composed not only of header information, but also the detail lines as well. The header and detail lines may each have public customer comments and private customer service comments attached. In an order entry system, detailed technical information about a product item appearing on a customer order line may be accessible as well. This complex object-called an order can be very naturally modeled using a **series of aggregations**. An order processing system can then be constructed to model very closely the **natural aggregations** occurring in the real world.

Aggregation is a concept that is used to express "part of" types of associations between objects. An aggregate is, a conceptually, an **extended object** viewed as a Unit by some operations, but it can actually be composed of **multiple objects**. One aggregate may contain multiple **whole-part** structures, each viewable as a distinct aggregate. Components may, or may not exist in their own right and they may, or may not appear in multiple aggregates. Also an aggregate's components may themselves have their own components.

Aggregation is a **special kind of association**, adding additional meaning in some situations. Two objects form an aggregate if they are tightly connected by a whole-part relationship. If the two objects are normally viewed as independent objects, their relationship is usually considered an association. **Grady Booch** suggests these tests to determine whether a relationship is an aggregation or not:

- Would you use the phrase "**part of**" to describe it?
- Are some operations **on the whole** automatically applied to its parts?
- Are some attribute values propagated from the **whole to all** or **some parts**?
- Is there an **intrinsic asymmetry** to the association, where one object class is **subordinate to the other**?

If your answer is yes to any of these questions, you have an aggregation. An **aggregation** is not the same as a **generalization**. Generalization relates distinct classes as a way of structuring the definition of a single object. Super class and subclass refer to properties of one object. A generalization defines objects as an instance of a super class and an instance of a subclass. It is composed of classes that describe an object (often referred to as a kind of relationship). Aggregation relates object instances: **one object that is part of another**. Aggregation hierarchies are composed of object occurrences that are each part of an **assembly** object (often called a **part of** relationship). A complex object hierarchy can consist of both **aggregations and generalizations**.

Composition: **A stronger form of aggregation** is called **composition**, which implies exclusive ownership of the part of classes by the whole class. This means that parts may be created after a composite is created, but such parts will be explicitly removed before the destruction of the composite. In UML, filled diamonds, as shown in *Figure 2*, indicate the composition relationship.

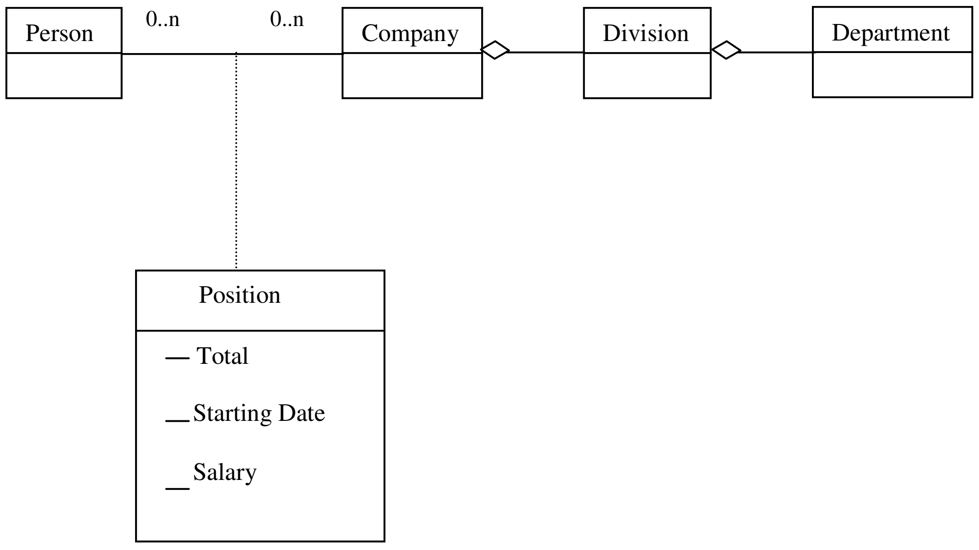

Figure 2: Example of a composition

Figure 2 shows that a person works for a company, company has many division, which are part of company and each division has many departments, which are again part of division.

1.2.2 Abstract Class

An abstract class is used to **specify the required behaviors** (operations, or method in java) of a class **without having to provide their actual implementations**. In other words you can say that methods without the implementation (body) are part of abstract classes.

An abstract object class has no occurrences. Objects of abstract classes are not created, but have child categories that contain the actual occurrences. A "concrete" class has actual occurrences of objects. If a class has at least one abstract method, it becomes, by definition, an abstract class, because it must have a subclass to override the abstract method. An abstract class must be sub classed; you cannot instantiate it directly.

Here you may ask one question: Why, are abstract classes are created? So the answer to this question is:

- To organise many specific subclasses with a super class that has no concrete use
- An abstract class can still have methods that are called by the subclasses. You have to take is this correct point in consideration in case of abstract classes
- An abstract method must be overridden in a subclass; you cannot call it directly
- Only a concrete class can appear at the bottom of a class hierarchy.

Another valuable question to ask is: why create the abstract class method? Answer to this question is:

- To contain functionality that will apply to many specific subclasses, but will have no concrete meaning in the super class.

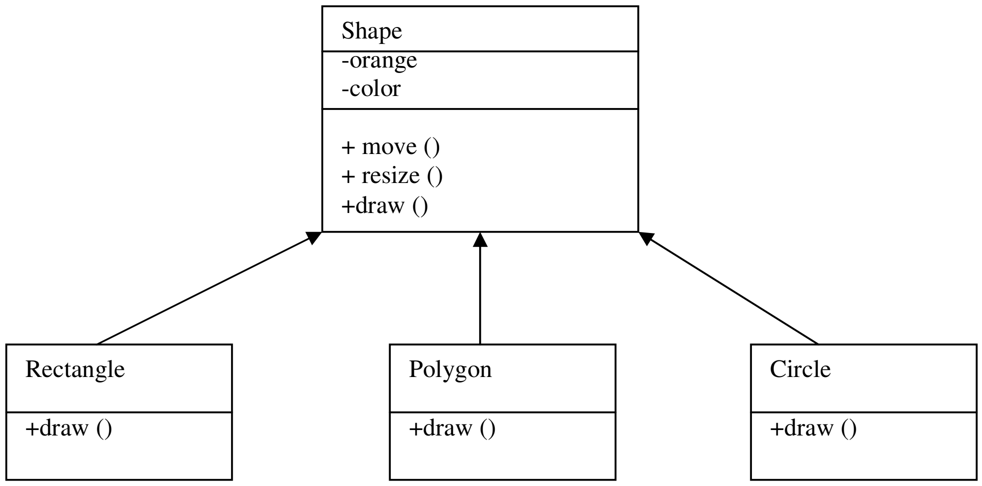

Figure 3: How abstract and concrete classes are related. Consider the example of shapes.

For example in *Figure 3*, you can see that the shape class is a natural super class for triangle, circle, etc.

Every shape requires a **draw () method**. But the method has no meaning in the Shape super class, so we make it abstract.

The subclasses provide the actual implementations of their draw methods since Rectangle, Polygon and Circle can be drawn in different ways. A subclass can override the implementation of an operation inherited from a super class by declaring another implementation. In this example, the draw method of rectangle class overrides the implementation of the draw operation inherited from the Shape class. The same applies to draw methods of Polygon and Circle.

Abstract classes can appear in the real world and can be created by modelers in order to promote **reuse of data and procedures** in systems. They are used to relate concepts that are common to multiple classes. An abstract class can be used to model an abstract super class in order to group classes that are associated with each other, or are aggregated together. An abstract class can define methods to be inherited by subclasses, or can define the **procedures for an operation** without defining a **corresponding method**. The abstract operation defines the pattern or an operation, which each concrete subclass must define in its implementation in their own way.

☞ **Check Your Progress 1**

Give the right choice for the followings:

1) A class inherits its parent's….

   (a) Attribute, links
   (b) Operations
   (c) Attributes, operations, relationships
   (d) Operations, relationships, link

   ……………………………………………………………………………

   ……………………………………………………………………………

2) If you wanted to organise elements into reusable groups with full information hiding you would use which one of the following UML constructs?

   (a) Package
   (b) Class
   (c) Class and interface
   (d) Sub-system or component

   ……………………………………………………………………………

   ……………………………………………………………………………

3) Which of the following is not characteristic of an object?

   (a) Identity
   (b) Behavior
   (c) Action
   (d) State

   ……………………………………………………………………………

   ……………………………………………………………………………

   ……………………………………………………………………………

4) Which of the following is not characteristic of an abstract class?

   (a) At least one abstract method
   (b) All the method have implementation (body)
   (c) Subclass must implement abstract method of super class
   (d) None of the above

   ……………………………………………………………………………

   ……………………………………………………………………………

Now, you are familiar with aggregation generalization, and abstract classes. As further extension of object oriented concepts, in the next section we will discuss multiple inheritance.

1.3 MULTIPLE INHERITANCE

Inheritance allows a class to inherit features from parent class (s). Inheritance allows you to create a new class from an existing class or existing classes.

Inheritance gives you several benefits by letting you:

- Reduce duplication by building on what you have created and debugged
- Organise your classes in ways that match the real world situations and entities.

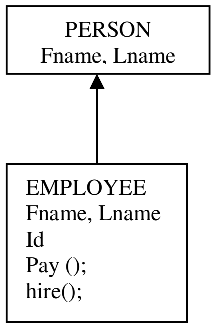

Figure 4: Example of Inheritance

For example, in *Figure 4*, you can see that EMPLOYEE class is inherited from PERSON class.

Multiple inheritance extends this concept to allow a class to have more than one parent class, and to inherit features form all parents. Thus, information may be mixed from **multiple sources**. It is a more complex kind of generalization; multiple inheritances does not restrict a class hierarchy to a tree structure (as you will find in single inheritance). Multiple inheritance provides greater modeling power for defining classes and enhances opportunities for reuse. By using multiple inheritance object models can more closely reflect the structure and function of the real world. The disadvantage of such models is that they become more complicated to understand and implement. See *Figure 5* for an example of multiple inheritance. In this example, the VAN classes has inherited properties from cargo Vehicle and Passenger vehicle.

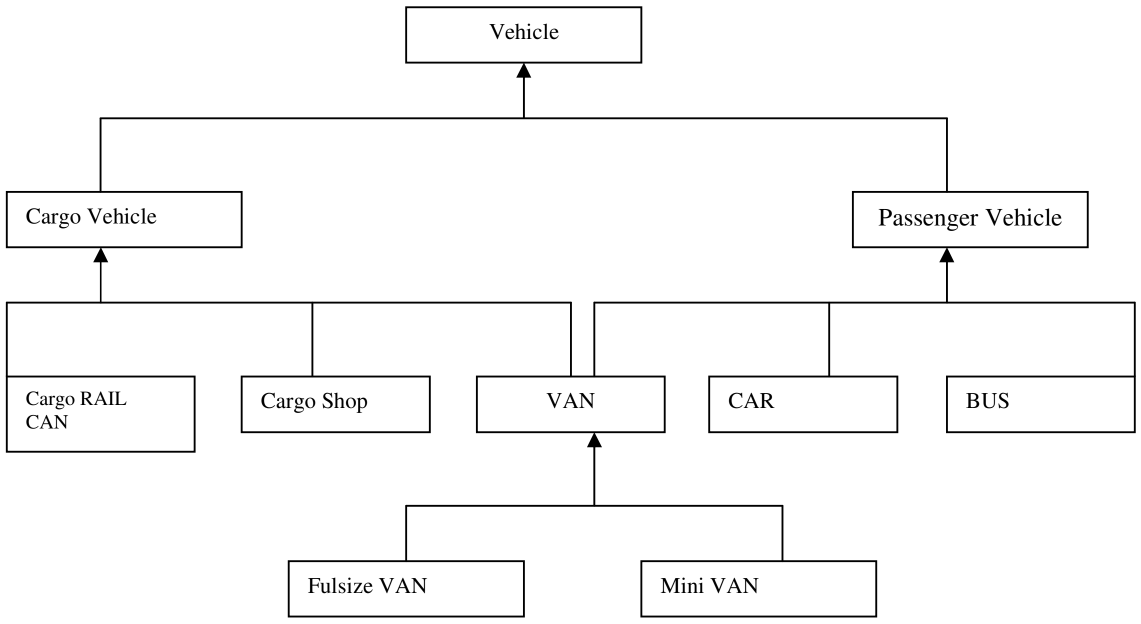

Figure 5: Example of Multiple Inheritance

The advantage of multiple inheritance is that it facilitates the re-use existing classes much better than single inheritance. If the properties of two existing classes have to be re-used and only single inheritance was available, then one of the two classes would have to be changed to become a subclass of the other class.

However, multiple inheritance should be used rather carefully. As the inheritance relationships that will be created through multiple inheritance may become rather complex and fairly difficult to understand. It is seen as a controversial aspect of object–orientation and, therefore, not implemented in some object-oriented languages, such as Smalltalk, because sometimes multiple inheritance can lead to ambiguous situations.

You will observe that:

- Working with multiple inheritance can be difficult in implementation if only single inheritance is supported, but analysis and design models can be restructured to provide a usable model. ***Rumbaugh et al***. discusses the use of delegation as an implementation mechanism by which an object can forward an operation to another object for execution. The recommended technique for restructuring-includes:

  Delegation using an aggregation of **roles a super class** with multiple independent generalizations can be recast as an aggregate in which each component replaces a generalization.

For this you have to:

- Inherit the most important class and delegate the rest. Here a join class is made a subclass of its most important super class.
- Nested generalization: factor on one generalization first, then the other, multiplying out all possible combinations.

Rumbaugh suggests issues to consider when selecting the best work around:

- If subclass has several super classes, all of equal importance, it may be best to use delegation, and preserve symmetry in the model
- If one super class clearly dominates and the others are less important, implementing multiple inheritance via single inheritance and delegation may be best
- If the number of combinations is small, consider nested generalization otherwise, avoid it
- If one super class has significantly more features than the other super classes, or one super class is clearly the performance bottleneck, preserve inheritance through this path
- If nested generalization is chosen, factor in the most important criterion first, then the next most important, etc.
- Try to avoid nested generalization if large quantities of code must be duplicated
- Consider the importance of maintaining strict identity (only nested generalization preserves this). Now, let us discuss the concept and specialization of generalization which is very important in respect of object oriented modeling.

1.4 GENERALIZATION AND SPECIALIZATION

Generalization means extracting common properties from a collection of classes, and placing them higher in the inheritance hierarchy, in a super class.

Generalization and **specialization** are the **reverse of** each other. An object type hierarchy that models generalization and specialization represents the most general concept at the top of an object type: hierarchy as the **parent** and the more specific object types as **children**.

Much care has to be taken when generalizing (as in the real world) that the property makes sense for **every single subclass** of **the super class**. If this is not the case, the property **must not** be generalized.

Specialization involves the definition of a new class which inherits all the characteristics of a **higher class** and **adds some new ones**, in a subclass.
Whether the creation of a particular class involves first, or second activity depends on the stage and state of analysis, whether initial classes suggested are **very general,** or **very particular**.

In other words, specialization is a **top-down** activity that refines the abstract class into more concrete classes, and generalization is a bottom-up activity that abstracts certain principles from existing classes, in order to find more abstract classes.

We often organise information in the real world as generalisation/***specialization*** hierarchies. You can see an example of generalization/specialization in *Figure 6*.

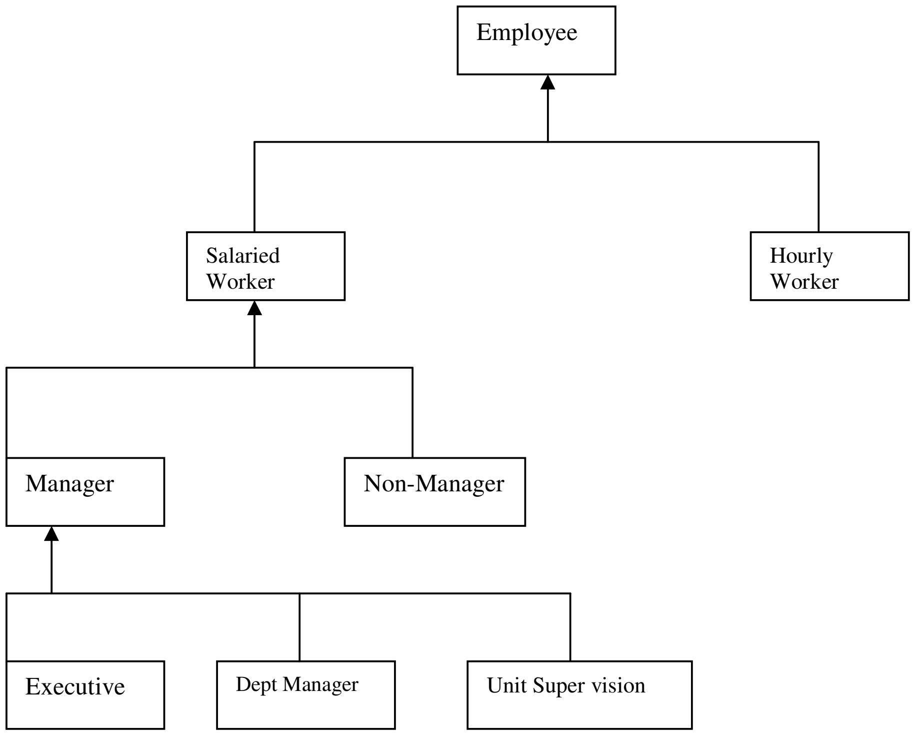

Figure 6: Generalization hierarchy of employee class

For instance, an employee may either be a salaried or an hourly worker. A salaried worker can be a manager, who in turn, can be an executive department manager, or a unit supervisor. The employee classification is **most general**, salaried worker is more **specific** and Unit supervisor is most specific. Of course, you might not model the world exactly like this for all organizations, but you get the idea. Unit Supervisor is a **subtype** of salaried worker, which is **a subtype of employee**. Employee is the highest-level super type and salaried worker is the super type of executive, department manager, and Unit supervisor. An object type could have several layers of subtypes and *subtypes of subtypes*. Generalization/specialization hierarchies help describe application systems and indicate where inheritance should be implemented in object-oriented programming language environments.

Any occurrence of a particular class is an occurrence of all ancestors of that class; so all features of a parent class automatically apply to subsclass occurrences. A child class cannot exclude, or suppress, an **attribute a parent**. Any operation on a parent must apply to all children. A child may modify an operation's implementation, but not is i**ts public** interface definition. A child class extends, parent features by adding new features. A child class may restrict the range of allowed values for inherited parent attributes.

In design and construction, operations on object data types can be over ridden, which **could substantially differ** from the original methods (rather than just refining original methods). Method overriding is performed to override **for extension, for restriction**, **for optimization,** or **for convenience**. Rumbaugh et al. proposes the following semantic rules for inheritance:

- All query operations (ones that read, but do not change, attribute values) are inherited by **all subclasses.**
- All update operations (ones that change attribute values) are inherited across **all extensions.**

- Update operations that change constrained attributes or associations are blocked across a restriction.
- Operations **may not** be overridden to make them **behave differently** in their externally visible manifestations from inherited operations. All methods that implement an operation must have the same protocol.
- Inherited operations can be refined by adding additional behavior.

Both generalization and specialization can lead to complex inheritance patterns, particularly via multiple inheritance. It is suggested that before making a final decision on generalisation/specialisation you should understand these rules very carefully and give the right choice for the following in respect of your system.

☞ **Check Your Progress 2**

1) Polymorphism can be described as

   (a) Hiding many different implementations behind one interface
   (b) Inheritance
   (c) Aggregation and association
   (d) Generalization

   ……………………………………………………………………………………

   …….………………………………………………………………………….

   ……………………………………………………………………………………

   ……………………………………………………………………………………

2) What phrase best represents a generalization relationship?

   (a) is a part of
   (b) is a kind of
   (c) is a replica of
   (d) is composed of

   ……………………………………………………………………………………

   …….……………………………………………………………………..…

   ……………………………………………………………………………………

   ……………………………………………………………………………………

3) All update operations in inheritance are updated

   (a) across all extensions
   (b) across only some of extensions
   (c) only first extension
   (d) None of the above.

   ……………………………………………………………………………………

   …….……………………………………………………………………..…

   ……………………………………………………………………………….…

1.5 METADATA AND KEYS

Now, we will discuss the basics of Metadata and keys.

Let us first discuss metadata. You are already familiar with metadata concept in your database course. As you know, RDBMS uses metadata for storing information of database tables. Basically, metadata is such set of data which describes other data. For example, if you have to describe an object, you must have a description of the class from which it is instantiated. Here, the data used to describe class will be treated as metadata. You may observe that every real-world thing may have meta data, because every real world thing has a description for them. Let us take the example of institutes and their directors. You can store that school A is having X as its direct, School B is having Y as its director, and so on. Now, you have concrete information to keep in metadata that is every institute is having a director.

KEY

Object instances may be identified by an attribute (or combination of attributes) called a key. A primary key is an attribute (or combination of attributes) that **uniquely identifies an object instance** and corresponds to the identifier of an actual object. For example, customer number would usually be used as the primary key for customer object instances. Two, or more, attributes in combination sometimes may be used to uniquely identify an object instance. For example, the combination of last name, first name and middle initial might be used to identify a customer or employee object instance. Here, you can say that sometimes more than one attribute gives a better chance to identify an object. For example, last name alone would not suffice because many people might have the same last name. First name would help but there is still a problem with uniqueness. All three parts of the name are better still, although a system generated customer or employee number is bests used as an identifier **if absolute uniqueness is desired**. Possible Primary Keys that are not actually selected and used as the primary keys are called **candidate keys**.

A **secondary key** is an attribute (or combination of attributes) that may **not uniquely identify an** object instance, but can describe a set of object instances that **share some common characteristic**. An attribute (customer type) might be used as a secondary key to group customers as internal to the business organisation (subsidiaries or divisions) or external to it. Many customers could be typed as internal or external at the same time, but the secondary key is useful to identify customers for pricing and customer service reasons.

1.6 INTEGRITY CONSTRAINTS

You have already studied integrity constraints in DBMS course. Here, we will review how these constraints are applied in the object oriented model.

Referential Integrity

Constraints on associations should be examined for referential integrity implications in the database models. Ask when referential integrity rules should be enforced. Immediately, or at a later time? When you are modeling object instances over time, you may need to introduce extra object classes to capture situations where attribute values can change over time. For instance, if you need to keep an audit trail of all changes to an order or invoice, you could add a date and time attribute to the order or invoice objects to allow for storage of a historical record of their instances. Each change to an instance would result in another instance of the object, stamped for data and time of instance creation.

Insert Rules

These rules determine the conditions under which a dependent class may be inserted and they deal with restrictions that the parent classes impose upon such insertions. The rules can be classified into six types.

Dependent	**:**	Permit insertion of child class instance only when the matching parent class instance already exists
Automatic	**:**	Always permit insertion of a child class instance. If the parent class instance does not exist, create one
Nullify	**:**	Always permit insertion of the child class instance.
Default	**:**	Always permit insertion of a child class in séance.
Customized	**:**	Allow child class instance insertion only if certain validity constraints are met.
No Effect	**:**	Always permit insertion of the child class instance. No matching parent class instances may or may not exist. No validity checking is performed.

Domain integrity

These integrity rules define constraints on valid values that attributes can assume. A domain is a set of valid values for a given attribute, a set of logical of conceptual values from which one or more attributes can draw their values. For example, India state codes might constitute the domain of attributes for employee state codes, customer state codes, and supplier state codes. Domain characteristics include such things as:

- Data type
- Data length
- Allowable value ranges
- Value uniqueness
- Whether a value can be null, or not.

Domain describes a valid set of values for an attribute, so that domain definitions can help you determine whether certain data manipulation operations make sense.

There are two ways to define domains.

One way to define domains and assign them to attribute is to define the domains first, and then to associate each attribute in your logical data model with a predefined domain. Another way is to assign domain characteristics to each attribute and then determine the domains by identifying certain similar groupings of domain characteristics. The first way seems better because it involves a thorough study of domain characteristics before assigning them to attributes. Domain definitions can be refined as you assign them to attributes. In practice, you may have to use the second method of defining domains due to characteristics of available repositories or CASE tools, which may not allow you to define a domain as a separate modeling construct.

Domain definitions are important because they:

- Verify that attribute values make business sense
- Determine whether two attribute occurrences of the same value really represent the same real-world value
- Determine whether various data manipulation operations make business sense.

A domain range characteristics for mortgage in a mortgage financing system could prevent a data entry clerk from entering an age of five years. Even though mortgage age and loan officer number can have the same data type, length and value, they definitely have different meanings and should not be related to each other in any data manipulation operations. The values 38 for age and 38 for officer number represent two **entirely unrelated values in the real world**, even though numerically they are exactly the same.

Table 1: Typical domain values

Data Characteristic	Example
Data type	character
	Integer
	Decimal
Data length	8 characters
	8 digits with 2 decimals
Allowable data values	x>=21
	0<x<100
Data value constraints	x in a set of allowable customer numbers
Uniqueness	x must be unique
Null values	x cannot be null
Default value	x can default the current date
	x can default to a dummy inventory tag number (for ordered items)

It makes little sense to match records based on values of age and loan officer number, even though it is possible. Matching customer in a customer class and customer payment in a customer transaction class makes a great deal of sense. Typical domain characteristics that can be associated with a class attribute are shown in *Table1***.**

Triggering Operation Integrity Rules

Triggering operation integrity rules govern **insert, delete, update** and **retrieval validity**. These rules involve the effects of operations on other classes, or on other attributes within class, and include domains, and insert/delete, and other attribute within a class, and include **domains** and **insert/delete** and other attributes business rules.

Triggering operation constraints involve:

- Attributes across multiple classes or instances
- Two or more attributes within a class
- One attribute or class and an external parameter.

Example triggering constraints include:

- An employee may only save up to three weeks time off
- A customer may not exceed a predetermined credit limit
- All customer invoices must include at least one line items
- Order dates must be current, or future dates.

Triggering operations have two components. These are:

- The event or condition that causes an operation to execute
- The action set in motion by the event or condition.

When you define triggering rules, you are concerned only with the logic of the operations, not execution efficiency, or the particular implementation of the rules. You will implement and tune the rule processing later when you translate the logical database model to a physical database implementation. Here, you should note that it is important to avoid defining processing solutions (like defining special attributes to serve as processing flags, such as a posted indicator for invoices) until all information requirements have been defined in the logical object model, and fully understood.

Triggering operations can be similar to referential integrity constraints, which focus on valid deletions of parent class instances and insertions of child class instances. Ask specific questions about each association, attribute, and class in order to elucidate necessary rules regarding data entry and processing constraints.

Triggering operation rules:

- Define rules for all attributes that are sources for other derived attributes
- Define rules for subtypes so that when a subtype instance is deleted, the matching super type is also deleted
- Define rules for item initiated integrity constraints.

1.7 AN OBJECT MODEL

In this last section of this unit let us examine a sales order system object model:

In this Sales Order System example, there are three methods of payment: **cash, credit Card or Check**. The attribute amount is common to all the three-payment methods, however, they have their own individual behaviors. *Figure 8* shows the object model, where the **directional association** in the diagram indicates the direction of navigation from one class to the other class.

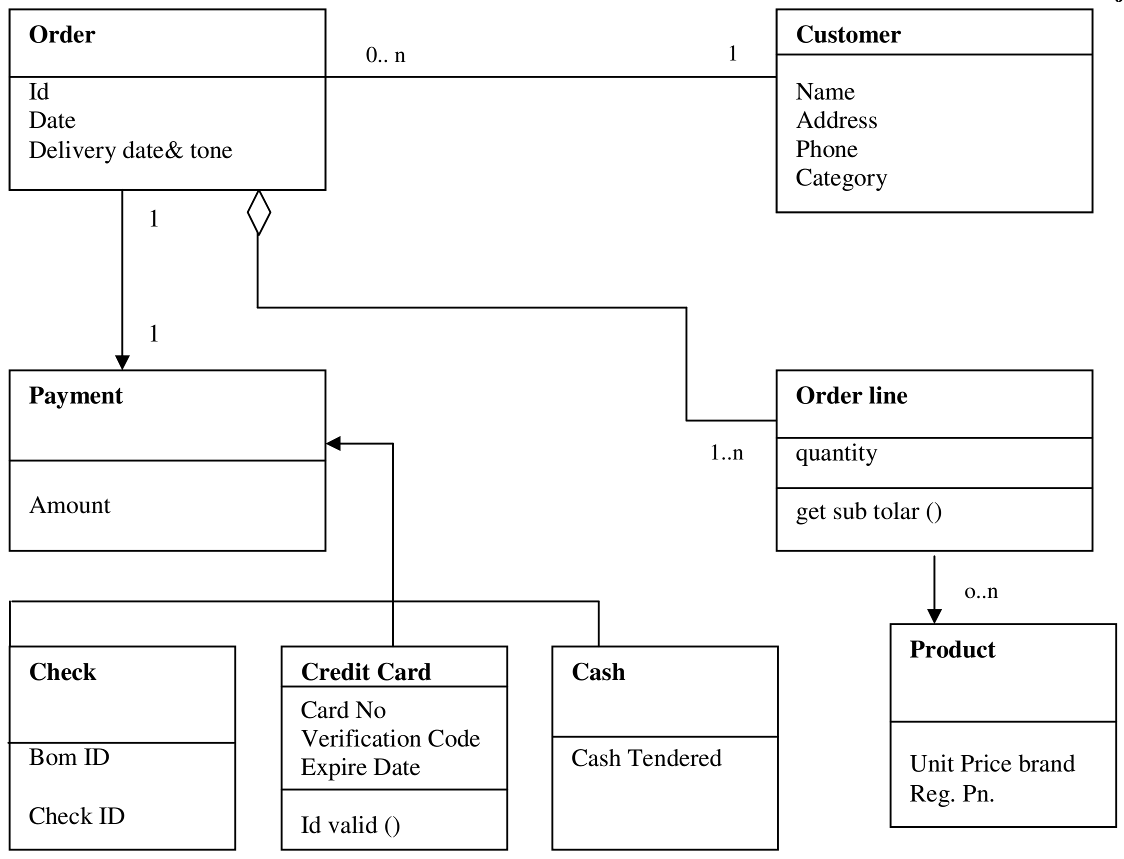

Figure 7: An object model for Sales Order System

☞ Check Your Progress 3

1) Explain in a few words whether the following UML class diagrams are correct or not

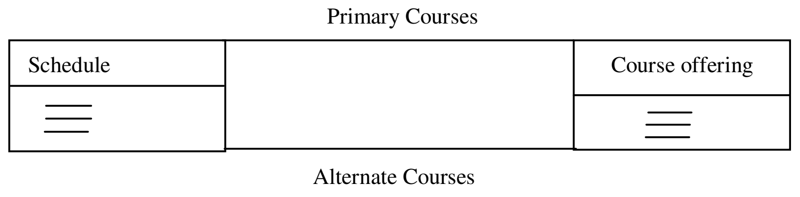

Figure 8: A Class Diagram

……………………………………………………………………………………………

……………………………………………………………………………………………

……………………………………………………………………………………………

2) Suppose that a computer is built out of one or more CPUs, Sound Card, and Video. Model the system with representative classes, and draw the class diagram

……………………………………………………………………………………………

……………………………………………………………………………………………

……………………………………………………………………………………………

……………………………………………………………………………………………

3) Suppose that Window WithScrollbar class is a subclass of Window and has a scrollbar. Draw the class diagram (relationship and multiplicity).

……………………………………………………………………………………

……………………………………………………………………………………

……………………………………………………………………………………

……………………………………………………………………………………

……………………………………………………………………………………

……………………………………………………………………………………

1.8 SUMMARY

This Unit over basic aspects of object modeling which includes discussion model based on objects (rather than functions) will be more stable over on a time hence the object oriented designs are more maintainable. Object do not exist in isolation from one another. In UML, there are different types of relationship (aggregation, composition, generalization). This Unit covers aggregation, and emphasizes that aggregation is the has-a or whole/part relationship. In this Unit we have also seen that generalization means extracting common properties from a collection of classes and placing them higher in the inheritance hierarchy, in a super class. We concluded the Unit with a discussion on integrity constraints, and by giving an object model for sales order system.

1.9 SOLUTIONS/ANSWERS

Check Your Progress 1

1) C 2) D 3) C 4) D

Check Your Progress 2

1) A 2) 2 3) B

Check Your Progress 3

1) **Incorrect**: Classes are not allowed to have multiple association, unless defined by different roles

2)

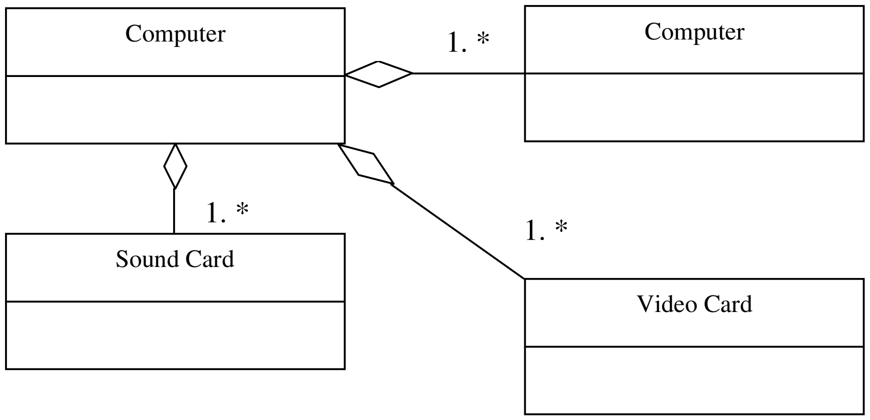

Figure 9: Class Diagram

3)

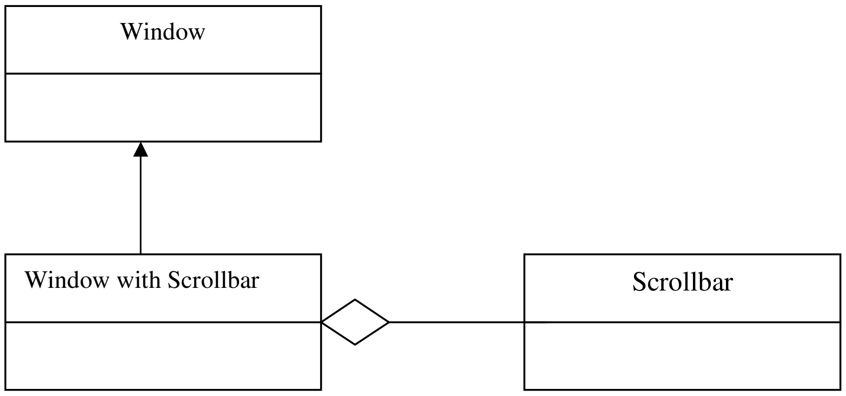

Figure 10: Class Diagram