UNIT 6 OBJECT MODELING

Structure **Page Nos.**

6.0 INTRODUCTION

In the previous Blocks of this course, we have learned the basics of OOA and OOAD. Also, we have seen how to draw and use different UML diagrams for visualizing, specifying, constructing, and documenting. According to Grady Booch, "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". Proper analysis of the system helps in identifying proper system requirements and finally in good designing of the system.

The goal of object design is object-oriented decomposition which leads to identifying the object that the system contains and the interactions between them. The system implements the specification. The goals of object-oriented design are:

(1) closer to the problem domain

(2) Incremental change is easy

(3) It supports reuse. Objects during Object-Oriented Analysis OOA focuses on a problem or, in other words, semantic objects. In Object-Oriented Design we focus on defining a solution. Object-Oriented modeling has 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, generalization in different forms and metadata.

In this unit, we will discuss Advanced Modeling Concepts, including, Aggregation & Abstract Class, Generalization & Specialization, Multiple Inheritance and designing of an Object Model

6.1 OBJECTIVES

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

- describe and apply the concept of aggregation,
- use the concepts of abstract Class, multiple Inheritance in designing systems,
- apply generalization as an extension,
- apply generalization as a Restriction, and
- explain the concept of metadata and constraints.

6.2 ADVANCED MODELING CONCEPTS

The object model describes the structure of the objects in a system.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 operations
5) optimize and package.

The first step for object-oriented design is object modeling. Before we go into details of object modeling, we should first 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. The goal of object model design is to capture those aspects/concepts of the underlying system that are important to represent the system in real-world in the context of the problem to be solved.

In this object modeling process, we first have to identify objects, then structures, attributes, associations, and finally services. The object models are represented using object diagrams containing objects and classes.

6.2.1 Aggregation

Aggregation represents the has-a or part-of relationship. Aggregation is tightly coupled form of association. An aggregation depicts a complex object as an assembly of many object components. Where these components are also objects, you may characterize a house in terms of its roof, floors, foundation, walls, rooms, windows, etc. A room may, in turn, be, composed of walls, ceiling, floor, windows, and doors, as represented in *Figure 6.1*. Hence a house can be seen as an aggregation of many objects. The decomposing of a complex object into its component objects can be extended until the desired level of detail is reached. In UML, a link is placed between the "whole "and "parts" classes, with a diamond head as shown in Figure-6.1, attached to the whole class to indicate the aggregation. In aggregation, multiplicity is specified at the end of the association for each part-of classes to indicate the constituent parts of the association.

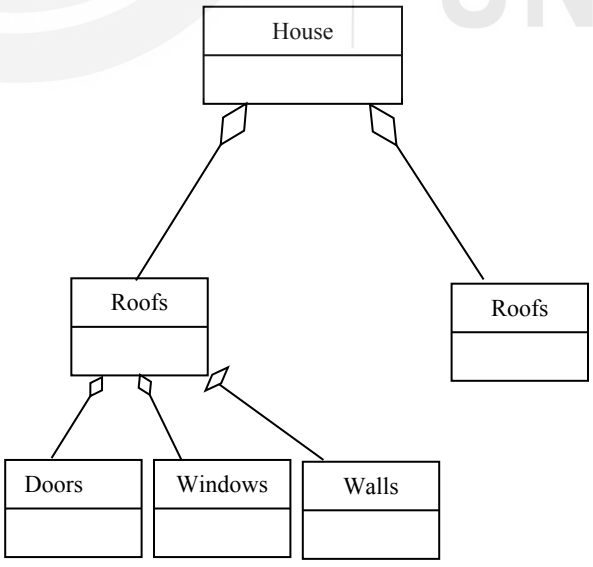


Figure 6.1: A house and some of its component

Object aggregation is very helpful in describing models of the **real-world** that are composed of other models. When describing a complex system of aggregates, System analysts need to describe them in enough detail for the system at hand. For example,

in the case of a customer order and services, a customer order is composed of header information and the detail lines. Also, the header and detail lines may each have attached public and private customer service comments. 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 modeled naturally using a **series of aggregations**. An order processing system can then be constructed to closely model the natural aggregations occurring in the real world.

As mentioned earlier, 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 single UNIT by some operations, but it can actually be composed of **multiple objects**. One aggregate in the system may contain multiple **whole-part** structures. Also, each part is viewable as a distinct aggregate. Components of the system may or may not exist in their own right and 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 a whole-part relationship tightly connects them. If the two objects are typically 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**?

You have an aggregation if your answer is yes to any of these questions. A**ggregation** is not the same as a **generalization**. Generalization relates distinct classes to structure the definition of a single object. Superclass and subclass refer to the properties of one object. A generalization defines objects as an instance of a superclass 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 part of an **assembly** object (often called a **part of** a relationship). A complex object hierarchy can consist of both **aggregations and generalizations**.

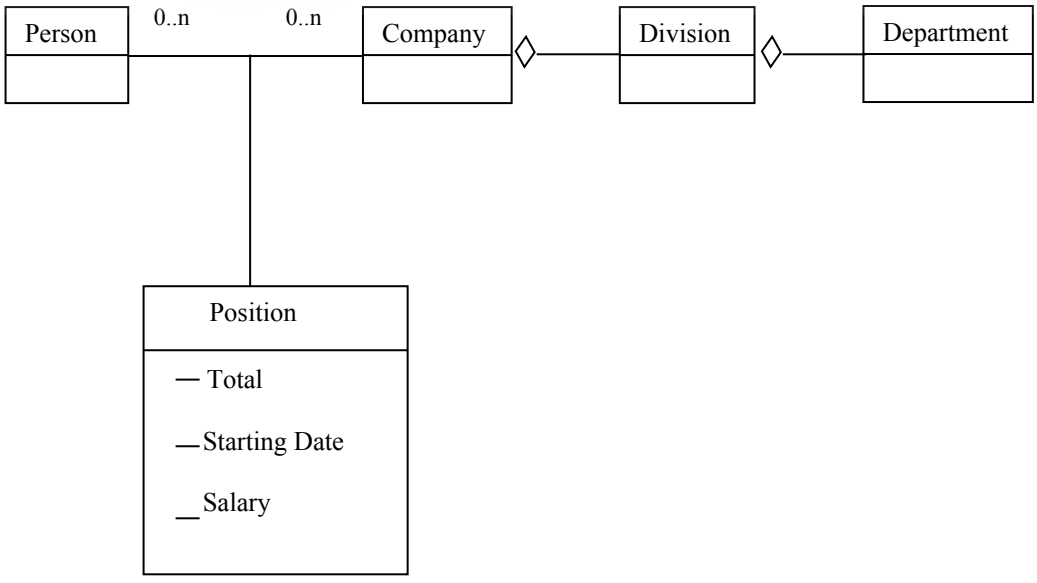


Figure 6.2: Example of a composition

Composition: **A stronger form of aggregation** is called **composition**, which implies exclusive ownership of the part of classes by the whole class. Composition means parts of the system may be created after a composite system is created. Remember that in such systems parts will be explicitly removed before the destruction of the composite. In UML, filled diamonds, as shown in *Figure 6.2*, indicate the composition relationship.

Figure 6.2 shows that a person works for a company, the company has many divisions, which are part of the company, and each division has many departments, which are again part of the division.

6.2.2 Abstract Class

An abstract class is used to **specify the required behaviors** (operations or methods) of a class **without providing their actual implementations**. In other words, you can say that methods without 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 containing 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 subclassed; you cannot instantiate it directly.

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

- To organize many specific subclasses with a superclass 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 into consideration in the case of abstract classes,
- An abstract method must be overridden in a subclass; you cannot call it directly, and
- Only a concrete class can appear at the bottom of a class hierarchy.

Another helpful question to ask is: why create the abstract class method? The answer to this question is:

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

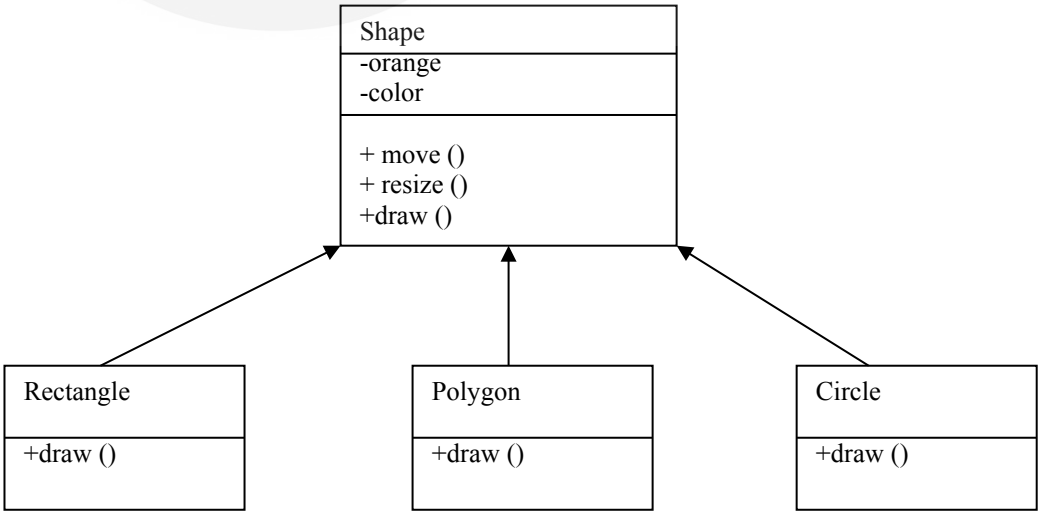


Figure 6.3: How abstract and concrete classes are related. Consider the example of shapes.

For example, in *Figure 6.3*, you can see that the shape class is a natural superclass for the shapes Rectangle, Polygon, Circle, etc.

Every shape requires a **draw () method**. But the method has not been provided with any meaning in the superclass Shape, so we make it an abstract class.

Subclasses provide another implementation to the method as per their need. The subclasses provide the implementations of their draw methods since Rectangle, Polygon and Circle can be drawn differently. A subclass overrides the implementation of an operation inherited from a superclass. In this example, the draw method of the Rectangle class overrides the implementation of the draw operation inherited from the Shape class. The same applies to the draw methods of Polygon and Circle.

Abstract classes can appear in the real world and be created by modelers to promote the **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 superclass to group classes that are associated with each other or aggregated together. An abstract class can define methods inherited by subclasses or 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 its own way.

☞ Check Your Progress 1

Give a 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) Which of the following is not characteristic of an object?

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

   ……
   ……
   ……

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

   (a) At least one abstract method
   (b) Methods may or may not have an implementation(body)
   (c) Subclass must implement the abstract methods of superclass
   (d) None of the above

   ……
   ……
   ……

4) When abstract classes are to be used in a system?

   ……
   ……
   ……

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

6.3 MULTIPLE INHERITANCE

Inheritance allows a class to inherit features from the parent class (s). Inheritance allows you to create a new class from one or more existing class(es)

Inheritance gives several benefits such as:

- Reduce duplication and increase reusability by building on what you have already created.
- Organize theee classes of the system in the ways it may appear in real-world situations in the problem domain.

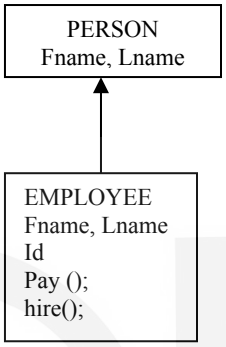


Figure 6.4: Example of Inheritance

For example, in *Figure 6.4*, you can see that the EMPLOYEE class is inherited from the PERSON class.

Multiple inheritance extends this concept to allow a class to have more than one parent class and inherit features from all parents. Thus, information may be mixed from **multiple sources**. It is a more complex form 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. 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. For an example of multiple inheritance, see *Figure 6.5*. In this example, the VAN classes have inherited properties from class Cargo Vehicle and class Passenger Vehicle.

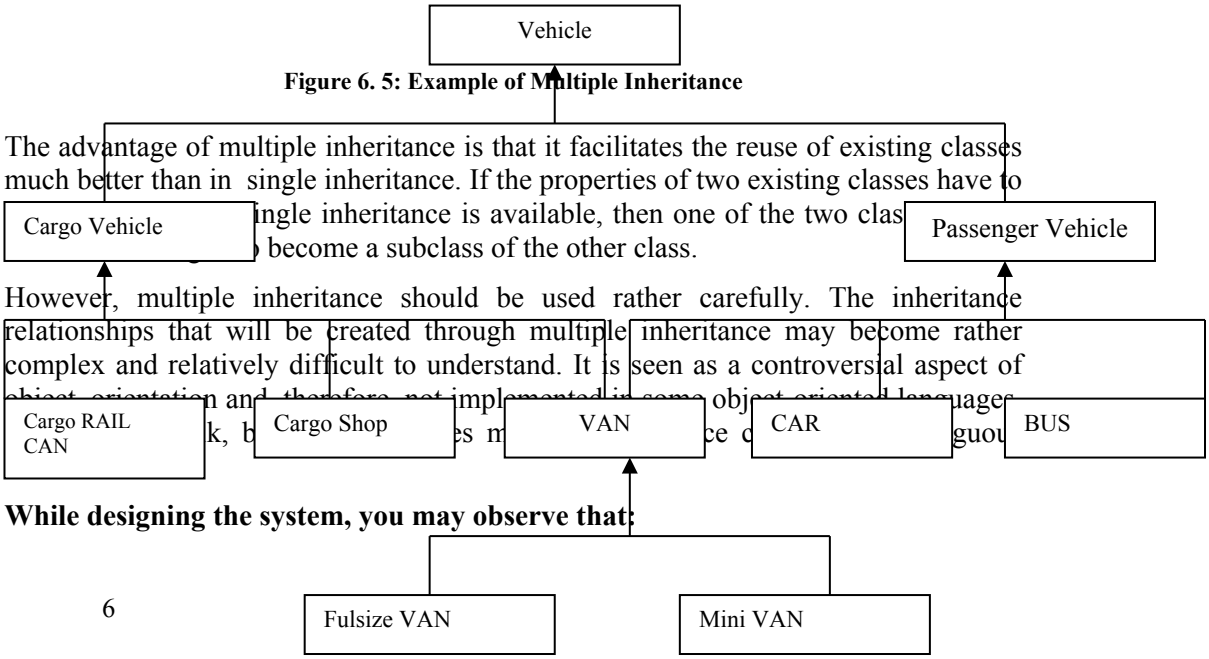


Figure 6. 5: Example of Multiple Inheritance

The advantage of multiple inheritance is that it facilitates the reuse of existing classes much better than in single inheritance. If the properties of two existing classes have to ingle inheritance is available, then one of the two clas become a subclass of the other class.

However, multiple inheritance should be used rather carefully. The inheritance relationships that will be created through multiple inheritance may become rather complex and relatively difficult to understand. It is seen as a controversial aspect of n and therefore not implemented in some object oriented languages k, b es m ce c guou

While designing the system, you may observe that:

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

  "Delegation using an aggregation of **roles, a superclass** 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 superclass.
- Nested generalization: factor on one generalization first, then the other, multiplying out all possible combinations.

Rumbaugh suggests issues to consider when selecting the best workaround:

- If the subclass has several superclasses, all of equal importance, it may be best to use delegation and preserve symmetry in the model
- If one superclass seems to dominate and the others are less important, implementing multiple inheritance via single inheritance and delegation may be best idea.
- If the number of combinations is small, consider nested generalization; otherwise, avoid it.
- If one superclass has significantly more features than the other superclasses or one superclass seems to have a performance bottleneck, preserve inheritance through this path.
- If nested generalization is chosen, factor in the most critical 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, an essential concept of object-oriented modeling.

6.4 GENERALIZATION AND SPECIALIZATION

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

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**.

When generalizing (as in the real world), care should be taken so that the property makes sense for **every single subclass** of **the superclass**. If this is not the case, the property **must not** be generalized.

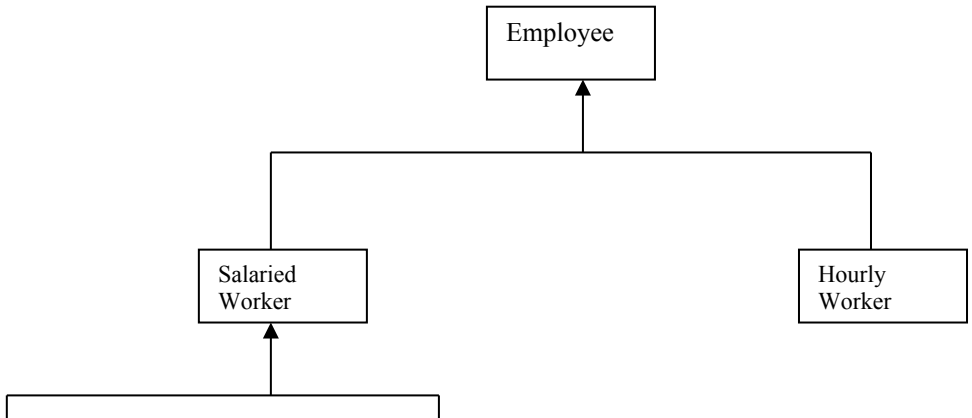


Figure 6.6: Generalization hierarchy of employee class

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 so that the derived class have some additional features compared to the superclass.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 to find more abstract classes.

We often organize information in the real world as generalization/specialization hierarchies.The creation of a particular class involves a first or second activity depends on the stage and state of analysis and whether the initial classes suggested are very general or very particular. You can see an example of generalization/specialization in Figure- 6.6.

For instance, an employee may either be a salaried or an hourly worker. A salaried worker can be a manager. Also, in turn, that employee can be an executive department manager or a unit supervisor. The employee classification is **most general**, the salaried worker is more **specific,** and also, the 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 the employee**. Employee is the highest-level supertype, and the salaried worker is the supertype 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 subclass occurrences. A child class cannot exclude or suppress an **attribute of a parent**. Any operation on a parent must apply to all children. A child may modify an operation's implementation but not 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 overridden, which **could substantially differ** from the original methods (rather than just refining original methods). Method overriding is performed to override **for extension, restriction, optimisation, or for convenience**. Rumbaugh et al. propose 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** from inherited operations in their externally visible manifestations. All methods that implement an operation must have the same protocol.
- Inherited operations can be refined by adding additional behaviour.

Both generalization and specialization can lead to complex inheritance patterns, mainly via multiple inheritance. It is suggested that before making a final decision on generalization/specialization you should understand these rules very carefully and make 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 the extensions
   (c) only first extension
   (d) None of the above.

   ……

   ……

   ……

4) Is generalisaaation and specialization reverse of each other? Why?

   ……

   ……

   ……

   In the next section, we will discuss the concept of metadata and keys in RDBMS.

6.5 METADATA AND KEYS

Let us first discuss metadata. As you know, RDBMS uses metadata for storing information of the database tables. Metadata can be defined as " such set of data which describes other data". For example, when you have to describe an object, you need to have a description of the class from which the object is instantiated. Here, the data you are using to describe class will be treated as metadata. You can say that every real-world thing may have metadata, because every real-world thing has a description for them. Let us take the example of an academic institutin and its directors. You can store that deparrrtttment A is having X as its director, and department B is having Y as its director, and so on. Now, you have concrete information to keep in metadata that every department in the institute has a director.

KEY

To identify an object, attribute or combination of attributes is used. This 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 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 of identifying an object. For example, last name alone would not suffice because many people might have the same last name. The 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 organization (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.

6.6 INTEGRITY CONSTRAINTS

You have already studied integrity constraints in the DBMS course. Here, we will have a look at 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 stage? 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 or 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.

The 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 that 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 specific 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 the 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 or not.
- Determine whether various data manipulation operations make business sense.

A domain range characteristics for a mortgage in a mortgage financing system could prevent a data entry clerk from entering the 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 they are numerically identical.

Table 6.1: 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 may be associated with some attributes of a class are shown in *Table-6.1*.

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 other attributes within the class, including domains, insert/delete, and other attributes within a class, including **domains** and **insert/delete** and other attributes business rules.

Triggering operation constraints involves:

- 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 of 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 essential 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 supertype is also deleted
- Define rules for item initiated integrity constraints.

6.7 AN OBJECT MODEL

In this section, 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 three payment methods. However, they have their own individual behaviours. *Figure- 6.7* shows the object model, where the **directional association** in the diagram indicates the direction of navigation from one class to the other class.

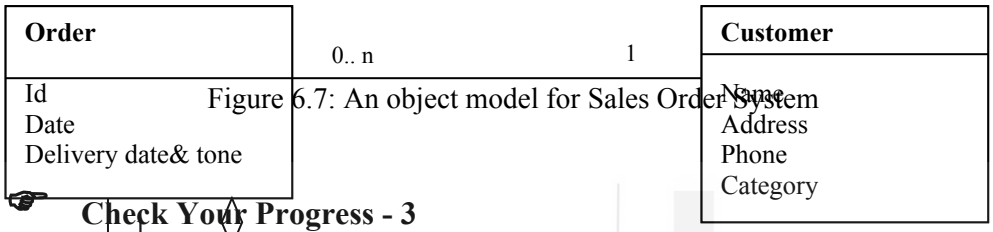


Figure 6.7: An object model for Sales Order System

Check Your Progress - 3

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

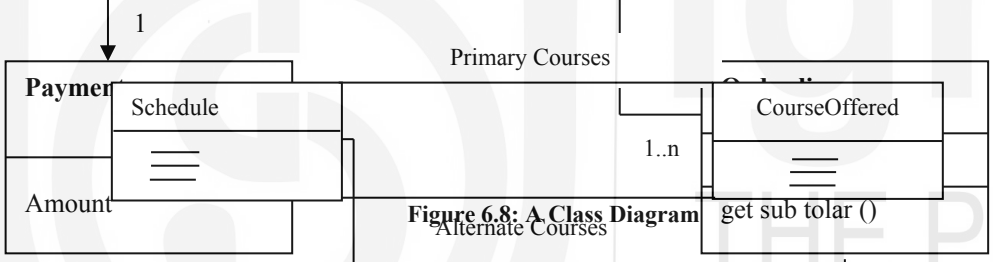
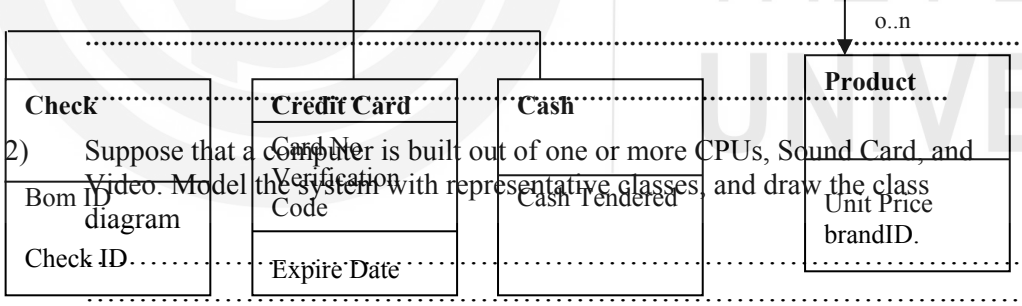


Figure 6.8: A Class Diagram



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 the WindowWithScrollbar class is a subclass of Window and has a scrollbar. Draw the class diagram (relationship and multiplicity).

6.8 SUMMARY

This unit explained basic aspects of object modeling, which includes a discussion on models based on objects (rather than functions) would be more stable over time hence the object-oriented designs are more maintainable. Objects do not exist in isolation

from one another. In UML, there are different types of relationships (aggregation, composition, generalization) which are used to represent object models. This unit covers aggregation and emphasizes that aggregation is the has-a or whole/part relationship. In this unit, we have also seen how using generalization extracting common properties can be performed from a collection of classes and placing them higher in the inheritance hierarchy in a superclass. The unit is concluded with a discussion on integrity constraints and, finally, by giving an object model for the sales order system.

6.9 SOLUTIONS/ANSWERS TO CHECK YOUR PROGRESS

Check Your Progress 1

1) c

2) c

3) d

4) Abstract class is used to relate concepts common to multiple classes while designing a system. It is to be used to model an abstract superclass to group classes associated with each other or aggregated together. Abstract class provides the opportunity to model system's common interface, which can be implemented differently by different subclasses in the system.

Check Your Progress - 2

1) a

2) b

3) a

4) Yes, g**eneralization** 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**. When the system is viewed from the children's place, the parent in the system looks like a very general form of the children.

Check Your Progress - 3

1) The diagram is incorrect. Classes are not allowed to have multiple associations unless defined by different roles.

2)

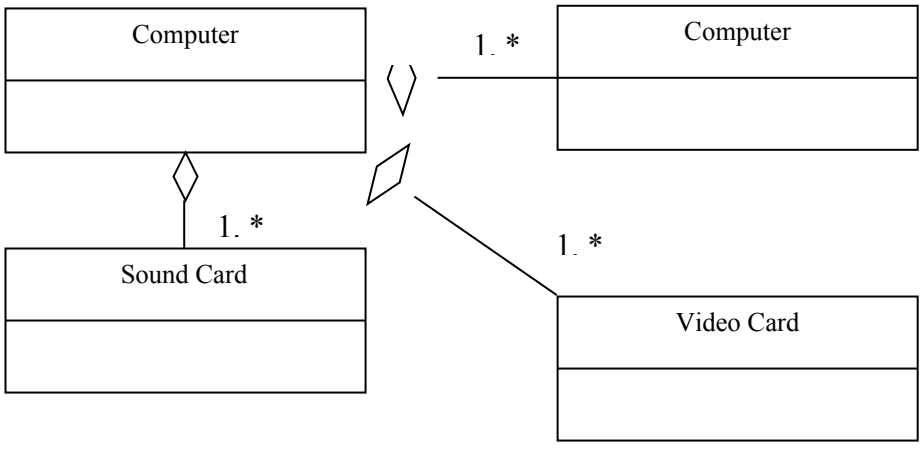


Figure 6.9: Class Diagram

3)

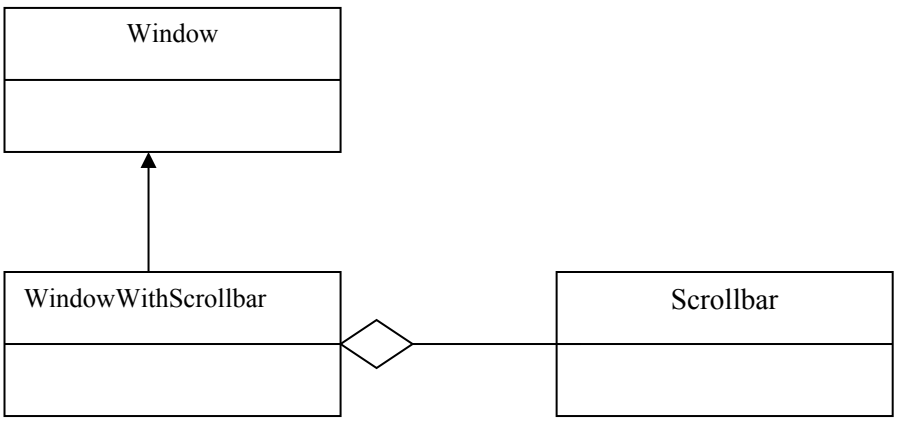


Figure 6.10: Class Diagram

6.10 REFERENCES/FURTHER READINGS

- Grady Booch, James Rumbaugh and Ivar Jacobson,“ The Unified Modeling Language User Guide”, 2nd Edition, Addison-Wesley Object Technology Series, 2005.
- Grady Booch, Robert A. Maksimchuk, Michael W. Engle, Bobbi J. Young, Jim Conallen, Kelli A. Houston, “Object-Oriented Analysis and Design with Applications,” 3rd Edition, Addison-Wesley, 2007.
- James Rumbaugh, Ivar Jacobson, and Grady Booch, “Unified Modeling Language Reference Manual,” 2nd Edition, Addison-Wesley Professional, 2004.
- John W. Satzinger, Robert B. Jackson, and Stephen D. Burd, “Object-oriented analysis and design with the Unified process,” 1st Edition, Cengage Learning India, 2007.