V SEMESTER: G 603.5: OBJECT ORIENTED ANALYSIS &DESIGN

UNIT - 1

- **Introduction**

  object orientation concept, OO development concept - Modelling concept, OO methodology, Three methods, OO Themes - Abstraction, Encapsulation, Combining data & behaviour, Sharing , Emphasis on the essence of an Object, Synergy

- **Modeling as a design Technique**

  Modeling, Abstraction, The 3 models

- **Class modeling**

  Object and class concepts - Objects, Classes, Class diagram, Values & attributes, Operation and methods, Link and Association Concepts - Link and association, Multiplicity, Association and names, Ordering, Bags & Sequences, Association Class, Qualified Association, Generalization and Inheritance- Definition, Use of generalization, Overriding features

- **Advanced Class Modeling**

  Multiplicity, Association Ends, Aggregation, Aggregation versus Association, Aggregation versus Composition

- **Case Study: A Sample class model**

UNIT – 2

- **State Modeling**

Events - Signal event, Change event, Time event, States, Transistors and conditions State Diagrams - Sample State Diagram, One shot state Diagrams, Summary of Basic state diagram notations, State Diagram Behavior - Activity Effects, Do Activities, Entry and Exit Activities, Completion Transition, Sending Signals

- **Interaction Modeling:**

Use Case Models

Actors, Use Cases, Use case Diagram, Guidelines for use case models

- **Sequence Model**: Scenarios, Sequence Diagram, ***Communication Diagram*****,** Activity Model - Activities, Branches, Introduction & termination, Concurrent Activities, Executable Activity diagram, Guidelines for Activity models, Deployment Diagram

- **Advanced Interaction modeling**

Use Case relationships- Include Relationships, Extend Relationship, Generalization, Combinations of use case relationships, Guidelines for use case relationships

Procedural Sequence Models- Sequence Diagrams with Passive Objects, Sequence Diagrams with Transient Objects, Guidelines for Procedural Sequence Models.

UNIT – 2

- **State Modeling**

Events - Signal event, Change event, Time event, States, Transistors and conditions State Diagrams - Sample State Diagram, One shot state Diagrams, Summary of Basic state diagram notations, State Diagram Behavior - Activity Effects, Do Activities, Entry and Exit Activities, Completion Transition, Sending Signals

- **Interaction Modeling:**

Use Case Models

Actors, Use Cases, Use case Diagram, Guidelines for use case models

- **Sequence Model**: Scenarios, Sequence Diagram, ***Communication Diagram*****,** Activity Model - Activities, Branches, Introduction & termination, Concurrent Activities, Executable Activity diagram, Guidelines for Activity models, Deployment Diagram

- **Advanced Interaction modeling**

Use Case relationships- Include Relationships, Extend Relationship, Generalization, Combinations of use case relationships, Guidelines for use case relationships

Procedural Sequence Models- Sequence Diagrams with Passive Objects, Sequence Diagrams with Transient Objects, Guidelines for Procedural Sequence Models.

UNIT - 3

- **Part 2: Analysis and Design**
- **Process Overview**

Development Stages - System Conception, Analysis, System Design, Class Design, Implementation, Testing, Training, Deployment, Maintenance

- **System Design** - Overview of System Design, Estimating Performance, Making a Reuse Plan - Library, Framework Pattern`Breaking a System into Sub-systems - Layers, Partitions, Combining Layers and Partitions, Identifying Concurrency - Identifying, inherent Concurrency, Defining Concurrent Tasks, Allocation of Sub-Systems - Estimating hardware Resource Requirement, Making Hardware and Software Trade-offs, Allocating Tasks to processors, Determining Physical Connectivity, Management of Data Storage, Handling Global Resources, Choosing a Software Controlled Strategy - Procedure Driven Control, Event Driven Control, Concurrent Driven Control, Internal Control, Other Paradigms, Handling Boundary Conditions, Setting Trade-off Priorities, Common Architectural Styles - Batch Transformation, Continuous Transformation, Interactive Interface, Dynamic Simulation, Real-time System, Transaction Manage

UNIT - 4

- **Class Design**

Overview of Class Design, Bridging the Gap, Realizing Use Cases, Designing Algorithms - Choosing Algorithms, Choosing Data structures, Defining Internal classes and Operations, Assigning Operations to Classes, Recursing Downward - Functionality Layers, Mechanism Layers, Refactoring, Design Optimization - Adding Redundant associations for Efficient Access, Saving derived values to avoid Re-computation, Rectification of Behavior, Adjustment of Inheritance - Rearranging Classes and Operations, Abstracting out Common Behavior, Using Delegation to share Behavior

- **Organizing a Class Design** - Information Hiding, Coherence of Entities, Fine Tuning Packages
- **Case Study - ATM, Library Management System (**Class Diagram, Object Diagram ,Use case Diagram ,Sequence Diagram, Collaboration Diagram ,State Diagram ,Activity Diagram ,Component Diagram ,Deployment Diagram)

- **Text Book**
- Object Oriented Modeling and Design with UML Michael R. Blaha James R. Rumbaugh, Second Edition, Pearson
- **Reference Books**
- UML™ 2 ToolKit – Hans-Erik Eriksson, Magnus Penker, Brian Lyons, David Fado, WILEY Publishing
- Object Oriented Analysis and Design with Applications Grady Booch Second Edition (Pearson Education)
- Object Oriented Software Engineering Bernd Brugge and Allen H. Dutoit Pearson Education

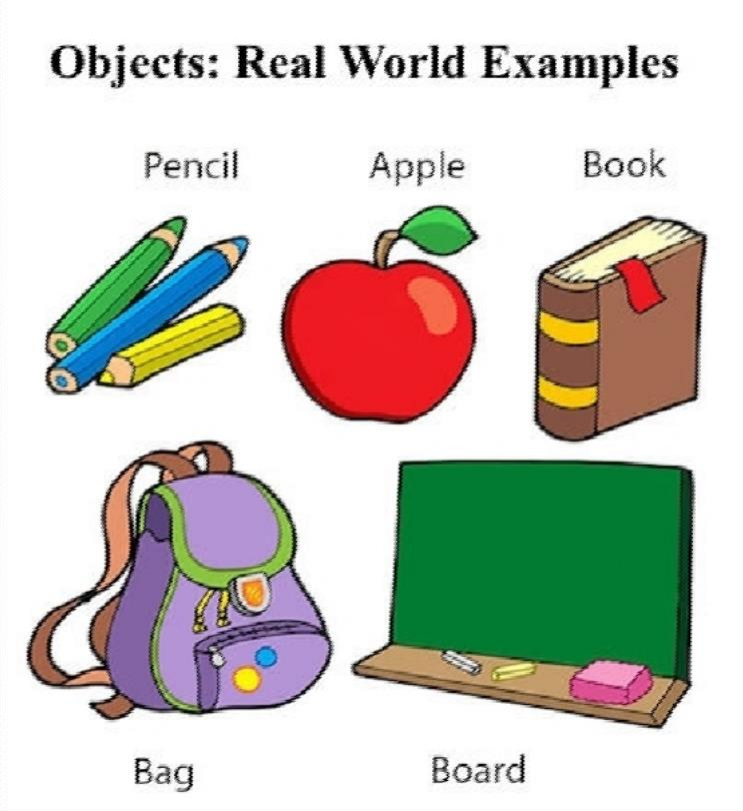
UNIT 1: CHAPTER 1
Introduction

Introduction to Object Orientation

What is object?

- object can be a variable, a data structure, a function, or a method, and as such, is
  a value in memory referenced by an identifier.
- Example

  dog, book, pencil, Apple, Bag, Board etc.



What is object orientation?

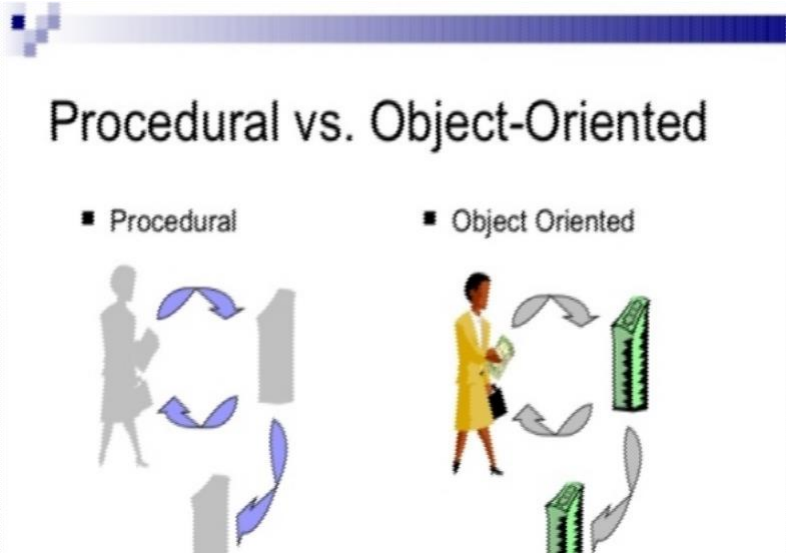
- Object orientation means that we organize software as a collection of discrete objects that incorporate both data structure and behaviour.
- Includes 4 aspects

  -Identity

  -Classification

  -Inheritance

  -Polymorphism



- **Identity** means that data is quantized into discrete, distinguishable entities calledobjects

  Example: customer, amount, account are the objects in bank transaction

- Each object has its own inherent identity

- **Classification** means that objects with the same data structure and behaviour are grouped into class.
- Each object is said to be an instance of its class

for example, in Bank transaction class Bank can have several instance of it.

- **Inheritance** is the sharing of attributes and features among classes based on a hierarchical relationship.
- A superclass has general information and subclasses refine and elaborate.
- For example, if Account is a base class then SB account or FD account can be its derived class .

- **Polymorphism** means that the same operation may behave differently for different classes

  For example, function display()

Object Oriented Development

- Object Oriented development refers to the software life cycle :

  analysis

  design

  implementation

- Object Oriented Development is the identification and organization of application concepts, rather than their final representation in a programming language.

- Modeling concepts

It focuses on front-end conceptual issues, rather than back-end implementation details

Benefits:

Helping specifiers, developers and customers express abstract concepts clearly and communicate them to each other.

It can serve as a medium for specification, analysis, documentation and interfacing.

- Object oriented methodology has the following stages

  System Conception

  Analysis

  System Design

  Class Design

  Implementation

➢ **System conception**

Software development begins with business analysts or users conceiving an application and formulating tentative requirements

Analysis

The analyst scrutinizes and restates the requirements from system conception by constructing models. The analyst must work with the requestor to understand the problem. The Analysis model briefly explains what the desired system must do, not how it will be done

- The analysis model has 2 parts

  - Domain Model

  - Application Model

  Domain Model: Description of real world objects reflected within the system

  Application Model: the description of the parts of the application system itself that are visible to the user

System Design

The system designer must decide what performance characteristics to optimize, choose a strategy of attacking the problem, and make tentative resource allocation.

For example, the system designer might decide that changes to the workstation screen must be fast and smooth, even when windows are moved or erased, and choose an appropriate communications protocol

Class Design

The focus of class design is the data structures and algoritnms needed to implement each class

Example :

The class designer determines data structures and algorithms for each of the operations of the window class

Implementation

Implementers translate the classes and relationships developed during class design into a particular programming language, database, orhardware.

During implementation, it is important to follow good software engineering practice so that traceability to the design is apparent and so that the system remains flexible and extensible.

Example:

Implementers would code the window class in a programming language.

- What is object?
- What is object orientation?
- 4 aspects of object orientation
- Stages of object orientation

- System Conception
- Analysis
- System Design
- Class Design
- Implementation

Three Models

- There are three kinds of models to describe a system from different viewpoints:
  - Class Model
  - State Model
  - Interaction Model

Class model

- Describes the static structure of the objects in a system and their relationships.
- Defines the context for software development
- The class model contains class diagrams
- Class diagram is a graph whose nodes are classes and whose arcs are relationship among classes

State Model

- State model describes the aspects of an object that change over time.
- This model specifies and implements control with state diagrams
- A state diagram is a graph whose nodes are states and whose arcs are transitions between states caused by events

Interaction Model

- Interaction Model describes how the objects in a system cooperate to achieve the result
- This model starts with **use cases** that are then elaborated with **sequence** and **activity** diagrams.

  - Use case focuses on the functionality of a system

  -sequence diagram shows the objects that interact and time sequence of their interaction

  -activity diagram elaborates important processing steps

Object Oriented Themes

- Abstraction
- Encapsulation
- Combining data and behaviour
- Sharing
- Emphasis on the Essence of an Object
- Synergy

- Abstraction

Abstraction lets you focus on essential aspects of an application while ignoring details. This means focusing on what an object is and does, before deciding how to implement it.

- Encapsulation

Encapsulation separates the external aspects of an object, that are accessible to other objects, from the internal implementation details, that are hidden from other objects.

Encapsulation prevents portions of a program from becoming so interdependent that small change has massive effects.

- Combining data and behaviour

The caller of an operation need not consider how many implementation exist.

Operator polymorphism shifts the burden of deciding what implementation to use from the calling code to the class hierarchy.

- Sharing

Object oriented techniques promote sharing at different levels. Inheritance of both data structure and behaviour lets subclasses share common code.

- Emphasis on the essence of an object

object oriented technology stresses what an object is, rather than how it is used. The use of an object depend on the details of the application and often change during the development

Object oriented development places a greater emphasis on data structure and a lesser emphasis on procedure structure

- Synergy

Identity, classification, polymorphism and inheritance characterize Object Oriented languages. Each of these concepts can be used in isolation but together they complement each other synergistically

UNIT 1: Chapter-2 Modelling as a design technique

Three Models

- The three kinds of models separate a system into distinct views.
- The class model represents the static, structural, “data” aspects of a system.
- The state model represents the temporal, behavioural, “control” aspects of a system
- The interaction model represents the collaboration of individual objects, the “interaction” aspects of a system

Class Model

- The class model describes the structure of objects in a system- identity, relationships to other objects, attributes and operations.
- The class model provides context for the state and interaction models
- Goal of constructing a class model is to capture those concepts from the real world that are important to an application

  - In modelling an engineering problem, the class model should contain terms familiar to engineers etc.

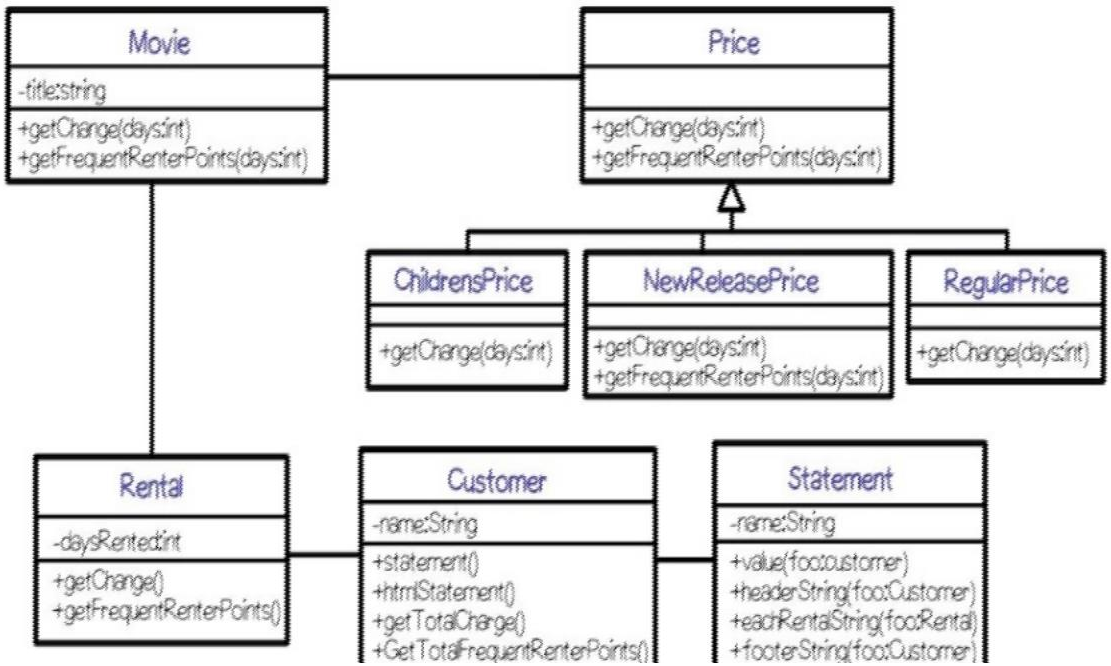
- Class diagrams express the classmodel.

Contd..

-Generalization lets classes share structure and behaviour and associations relate the classes

-Classes define the attribute values carried by each object and the operations that each object performs

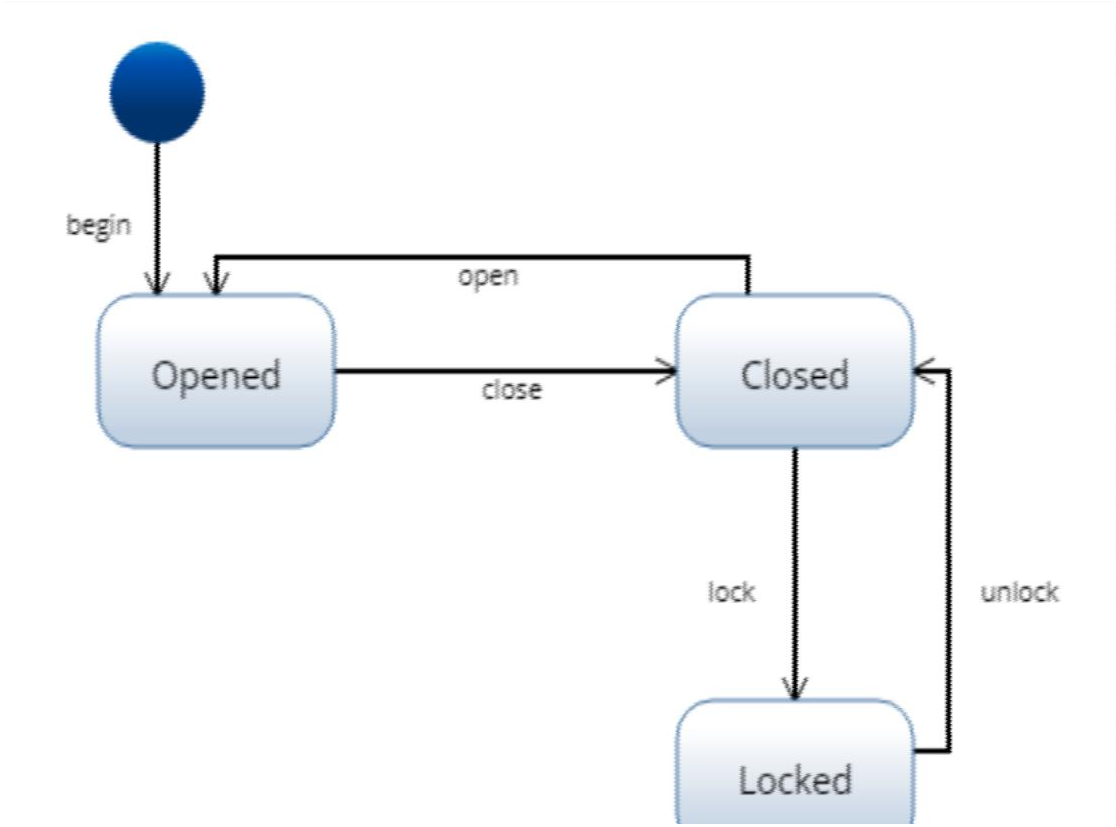
Example Class Diagram



State Model

- The state model describes those aspects of objects concerned with time and sequencing of operations
- The state model captures control, the aspects of a system that describes the sequences of operations that occur
- State diagrams express the state model.
- Each state diagram shows the state and event sequences permitted in a system for one class of objects

Contd..


begin
open
Opened
close
Closed
lock
unlock
Locked

Interaction Model

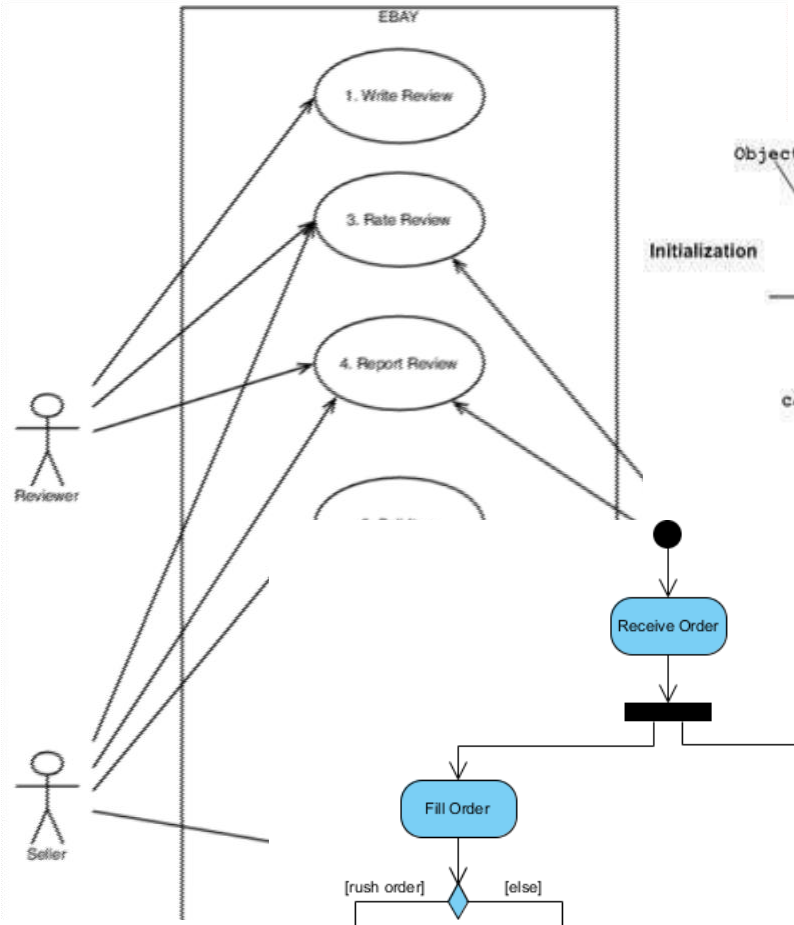
- The interaction model describes interactions between objects
- The state and interaction models describe different aspects of behaviour
- Use cases, sequence diagrams and activity diagrams document the interaction model

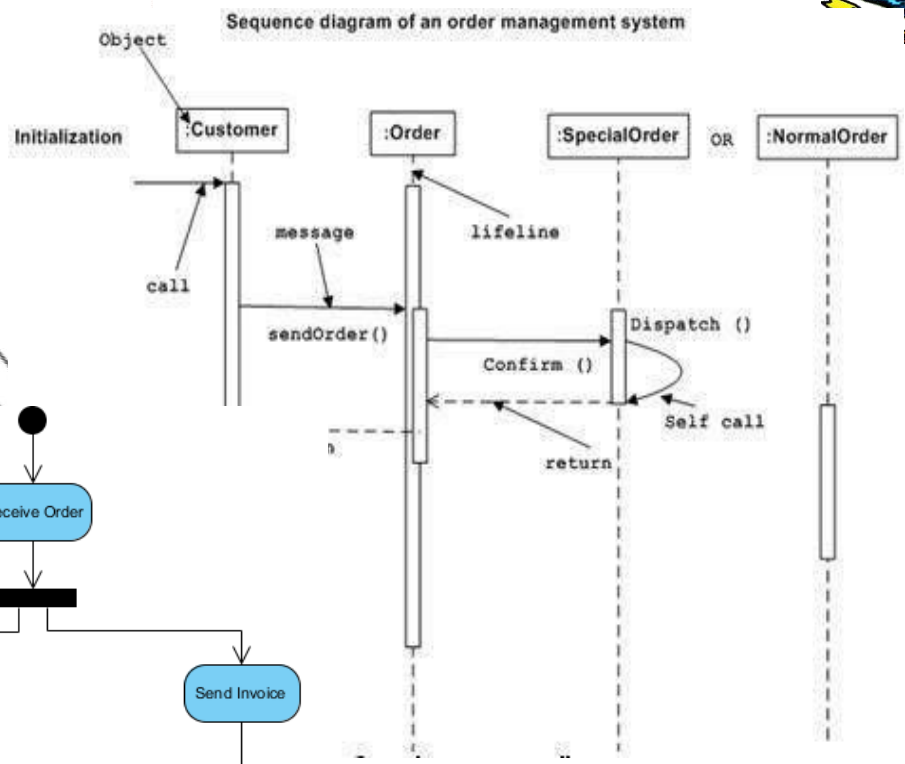
-use cases document major themes for interaction between the system and outside actors

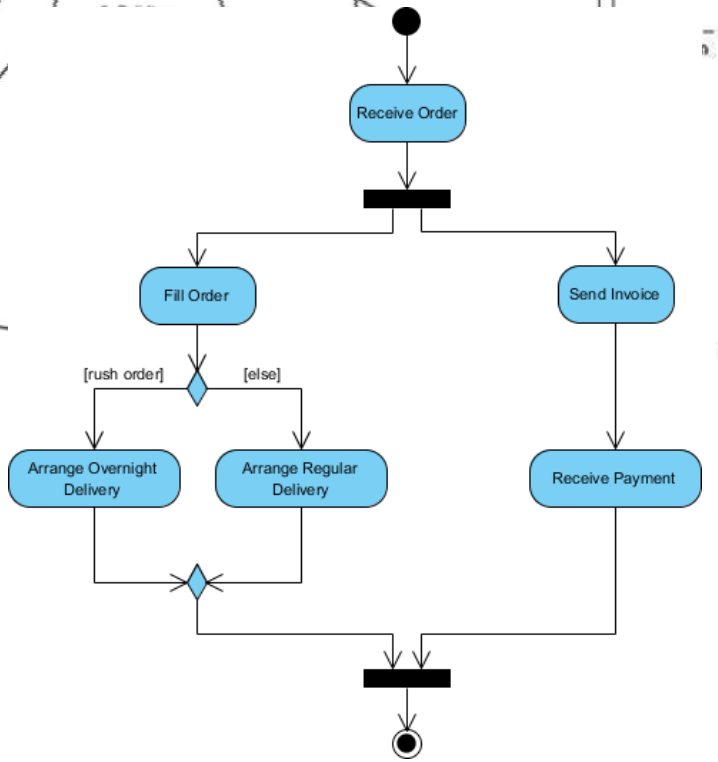
-sequence diagrams show the objects that interact and time sequence of their interactions

-Activity diagrams show the flow of control among the processing steps of a compuattion

Contd..


EBAY
1. Write Review
3. Rate Review
4. Report Review
Reviewer
Seller


Sequence diagram of an order management system
Object
Initialization
:Customer
:Order
:SpecialOrder
OR
:NormalOrder
call
message
lifeline
sendOrder()
Dispatch ()
Confirm ()
Self call
return


Receive Order
Fill Order
Send Invoice
[rush order]
[else]
Arrange Overnight Delivery
Arrange Regular Delivery
Receive Payment

Relationship among the Models

- Each model describes one aspect of the system but contains references to other models.
- The class model describes data structure on which the state and interaction models operate.
- the state model describes the control structure of objects
- The interaction model focuses on the exchanges between objects and provides a overview of the operation of a system

UNIT 1:Chapter 3 Class Modelling

contents

- Object and Class concepts

- - Objects, Classes, Class diagram, Values & attributes, Operation and methods,

- Link and Association concepts

- - Link and association, Multiplicity, Association and names, Ordering, Bags & Sequences, Association Class, Qualified Association

- Generalization and Inheritance

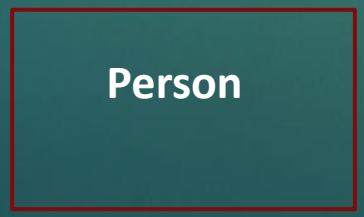
  - - Definition, Use of generalization, Overriding features

Object and Class Concepts

- Objects
- The purpose of class modelling is to describe objects.
- An object is a concept or thing with identity that has meaning for an application
  - *Example, Joe Smith, process number 7648*
- Some objects have real-world counterparts while others are conceptual entities
- All objects have identity and are distinguishable
- *Example, two apples with the same color, shape, and texture are still individual apples*

Class Diagrams

- There are two kinds of models of structure- class diagram and object diagram
- **Class diagrams** provide a graphic notation for modelling classes and their relationships.
- Class diagrams are useful for abstract modelling and for designing actual programs.
- The UML symbol for a class is box.
- Class name is written in boldface, center the name in the box, and capitalize the first letter



Object Diagram

- An object diagram shows individual objects and their relationships.
- **Object diagrams** are helpful for documenting test cases and discussing examples
- UML symbol for an object is a box with an object name and followed by a colon and a class name
- The object name and class name are both underlined.

Values and Attributes

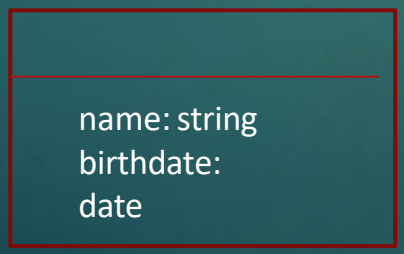
- An attribute is a named property of a class that describes a value

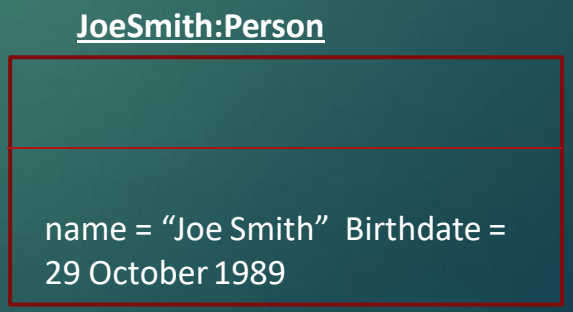
  For example: Name, birthdate, and weight are attributes of Person object

- A value is a piece of data

  For example: birthdate has a value "29 October 1989" for object JoeSmith

- The UML notation lists attributes in the second compartment of the class box.





Operations and Methods

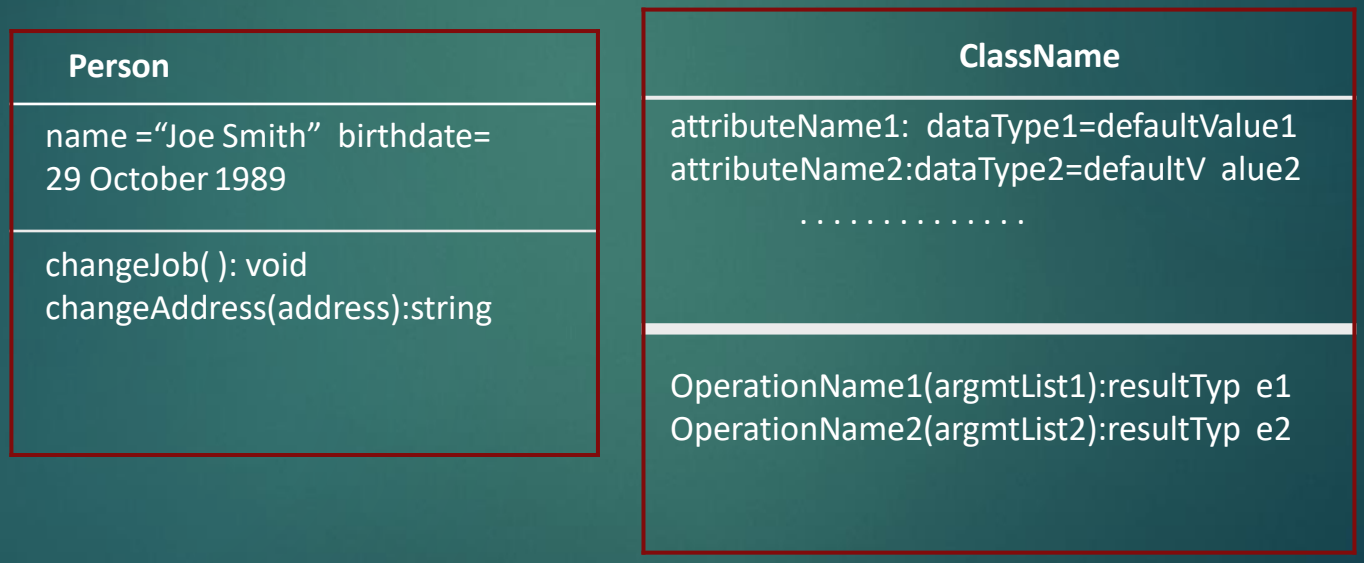
- An object is a function or procedure that may be applied to or by objects in a class.

  For example, Open, close, hide and redisplay are operations on class Window

- Each operation has a target object as an implicit argument
- The operation may apply to many different classes. Such an operation is polymorphic.

  For example, the class File may have an operation print

- Methods all should have the same signature- the number and types of arguments and the type of result value
- The UML notation is to list operations in the third compartment of the class box


Person
name ="Joe Smith" birthdate=
29 October 1989
changeJob( ): void
changeAddress(address):string
ClassName
attributeName1: dataType1=defaultValue1
attributeName2:dataType2=defaultV alue2
. . . . . .
OperationName1(argmtList1):resultTyp e1
OperationName2(argmtList2):resultTyp e2

Link and Association Concept

Links

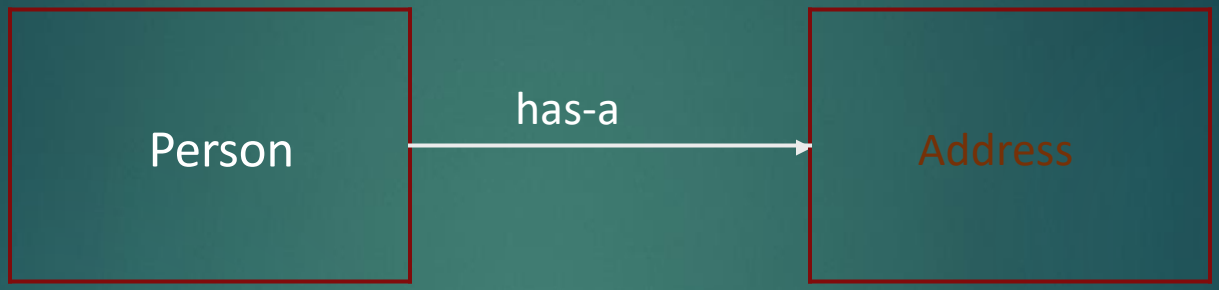
- A **link** is a physical or conceptual connection among objects
- Most links relate two objects, but some links relate three or more objects
- A link is an instance of an association
- The UML notation for a link is a line between objects

Association

- An **association** is a description of a group of links with common structure and common semantics
- Association show relationship between classes.
- Links and Associations often appear as verbs in problem statements.

Unary Association

- Class A knows about class B
- But class B knows nothing about class A

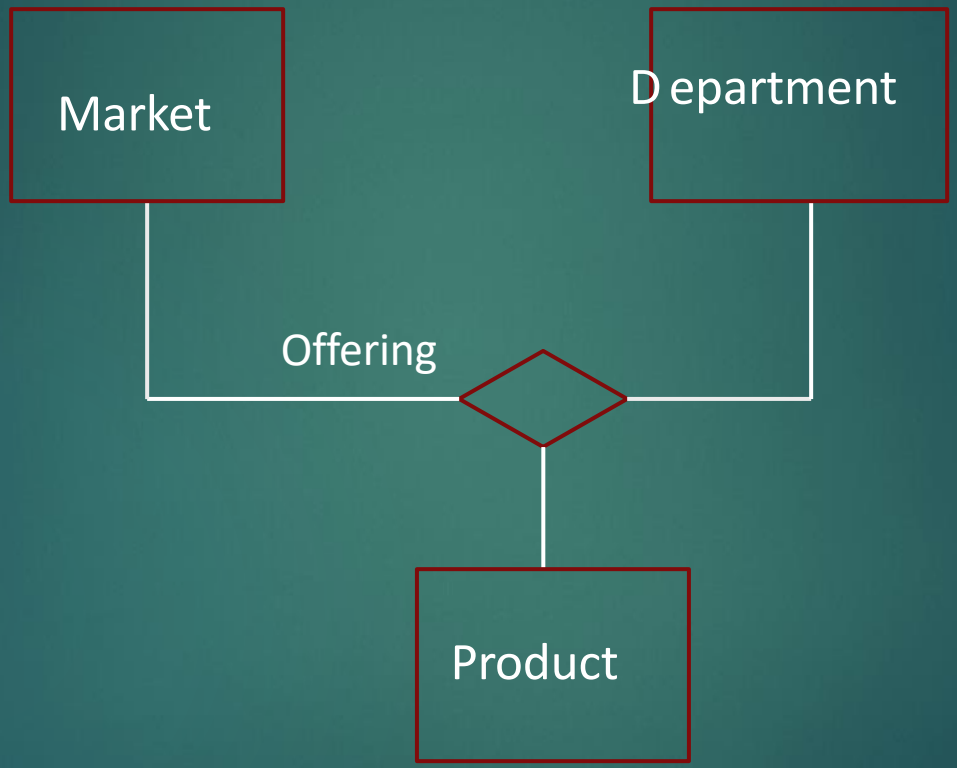

Person
has-a
Address

Binary Association

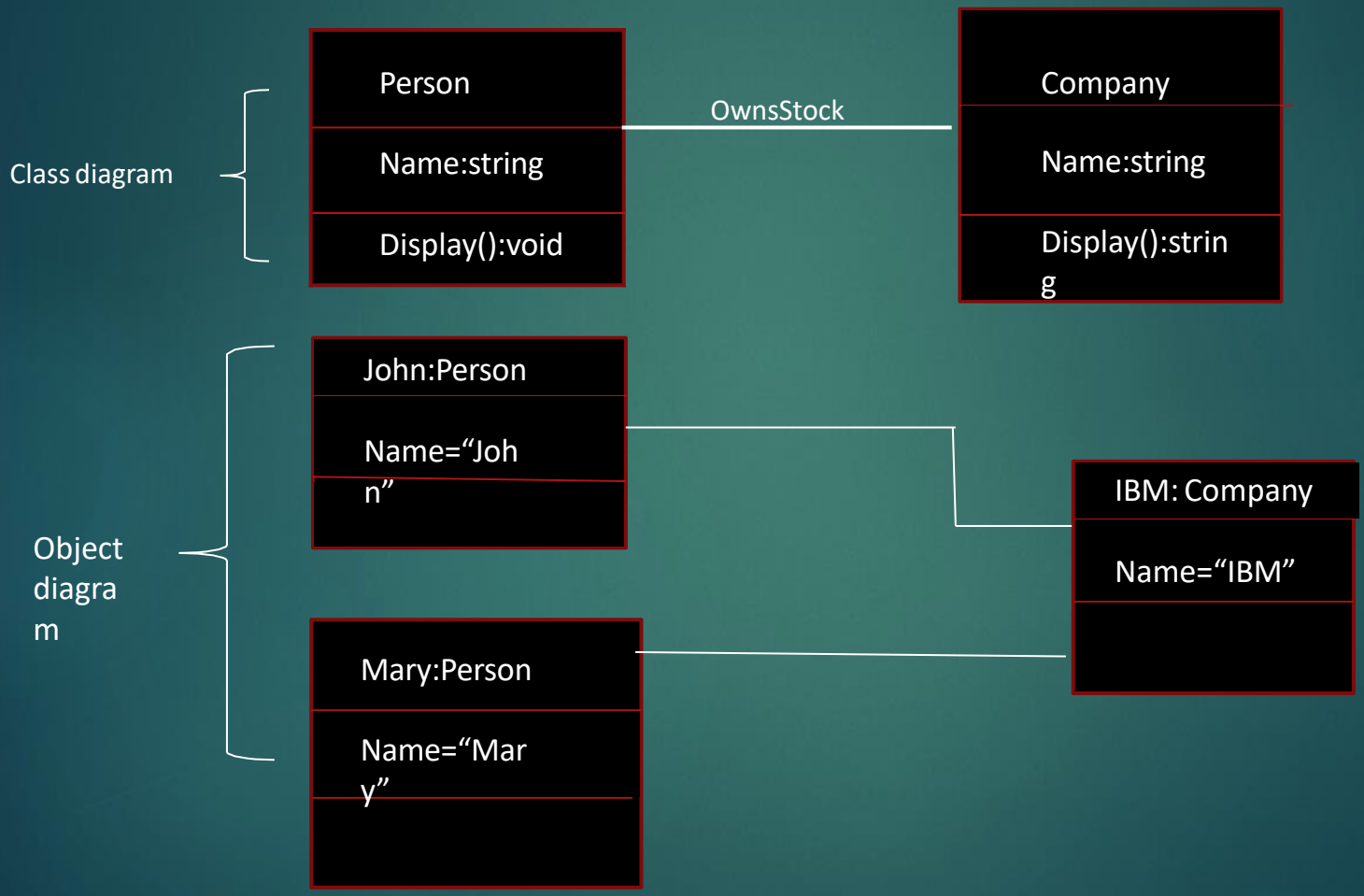
- Class C knows about class D
- Class D also knows about class C


Person
worksFor
Company

Ternary Association



For example, Joe Smith works-for IBM company



- The **association** name is optional, if the model is unambiguous.
- When there are multiple associations , must use association names or association end names to solve the ambiguity.

For example, *person works for company and person owns stock in company*

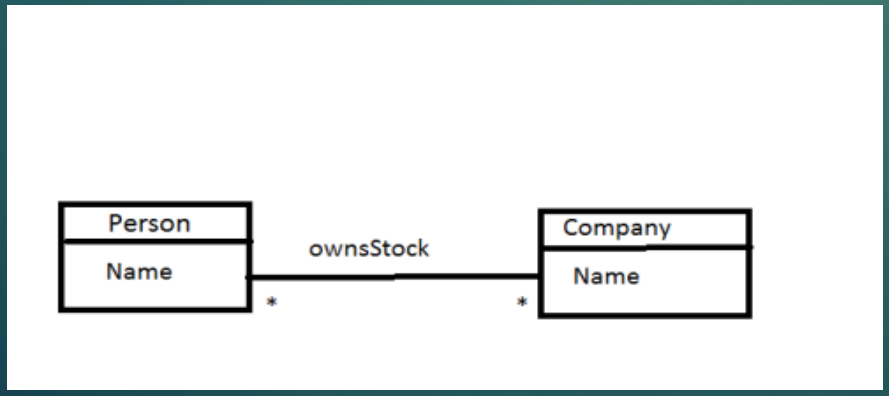
- Associations are bidirectional

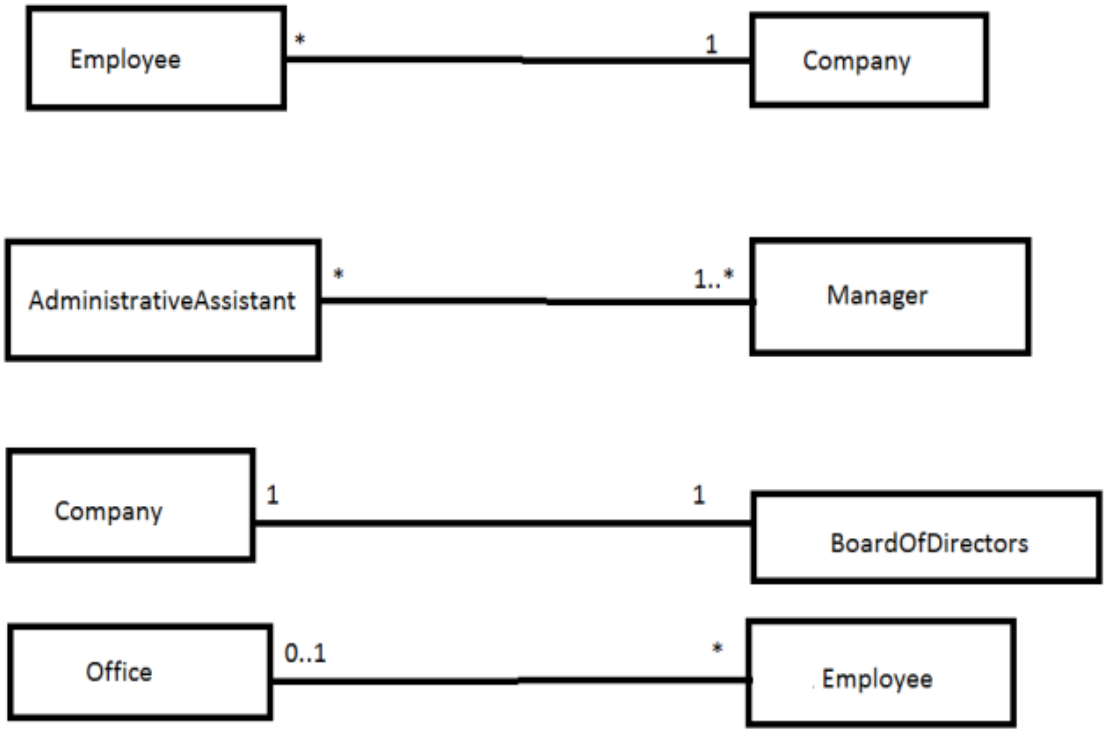
For example, ***WorksFor*** *connects a person to company. The inverse of WorksFor could be called* ***employs*** *and it connects a company to person*

Multiplicity

- Multiplicity specifies the number of instances of one class that may relate to a single instance of an associated class.
- UML diagrams explicitly list multiplicity at the ends of the association lines.
- The UML specifies multiplicity with an interval, such as "1" (exactly one), "1..*" (one or more), or "3..5" (three to five, inclusive) and "*" that denotes many.

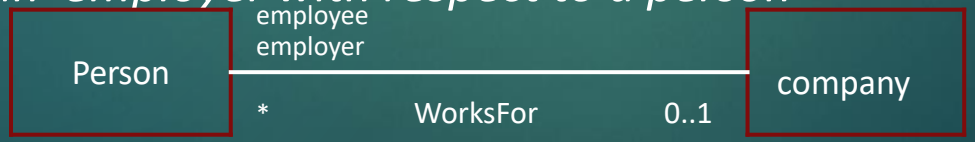
For example*, A person may own stock in many companies, A company may have multiple persons holding its stock*



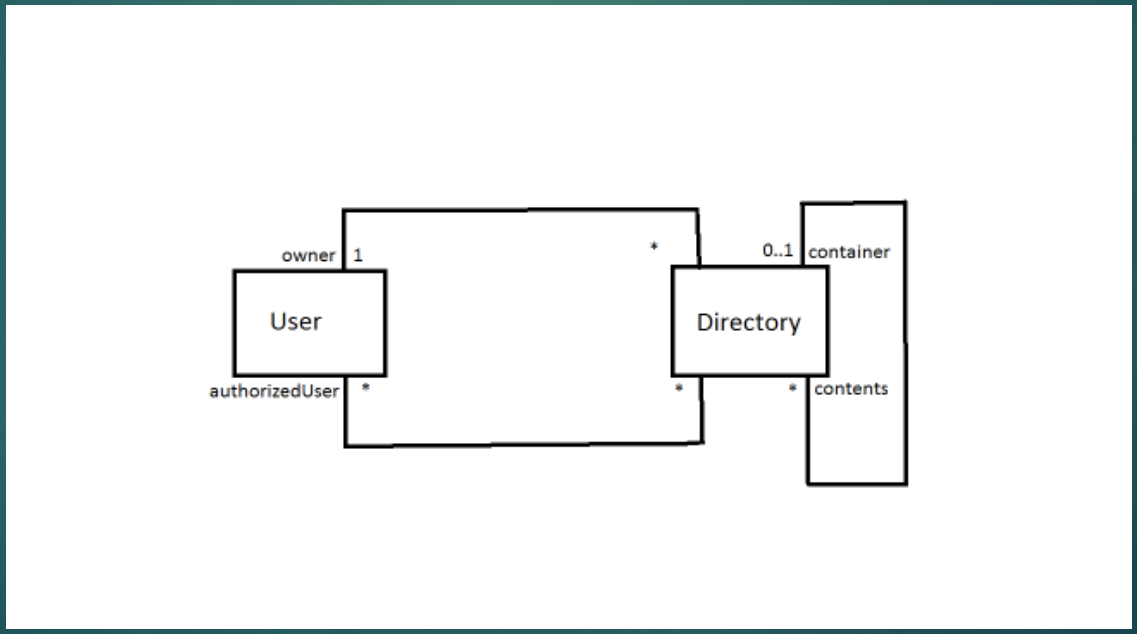

Employee
*
1
Company
AdministrativeAssistant
*
1..*
Manager
Company
1
1
BoardOfDirectors
Office
0..1
*
Employee

Association End Names

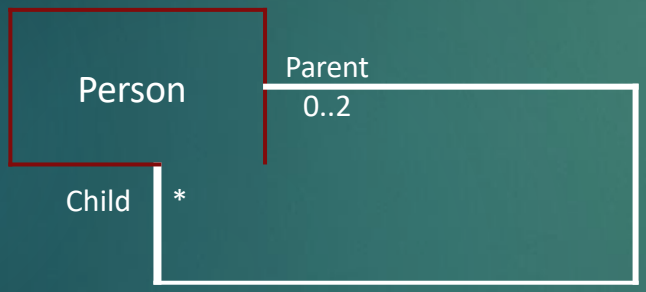
- Along with the multiplicity of association we can give names to the ends of association.
- Association end names often appear as nouns in problem description.
- Use of association end names is optional, but it is often easier and less confusing to assign association end names.
- For example, *in the following figure* ***Person*** *and* ***Company*** *participates in association WorksFor. Aperson is an employee with respect to a company; a company is an employer with respect to a person*



- Association end names are necessary for associations between two objects of the same class
- For example, container and contents distinguish the two usages of Directory in the self-association



- Association end names let you unify multiple references to the same class



Correct Model

Wrong Model

Ordering

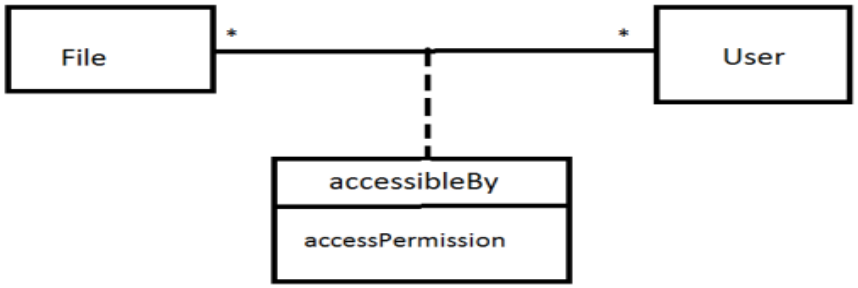
- Ordering or ordered set means there is an order between the elements at the end of the association
- We can indicate an ordered set of objects by writing “{ordered}” next to appropriate association end

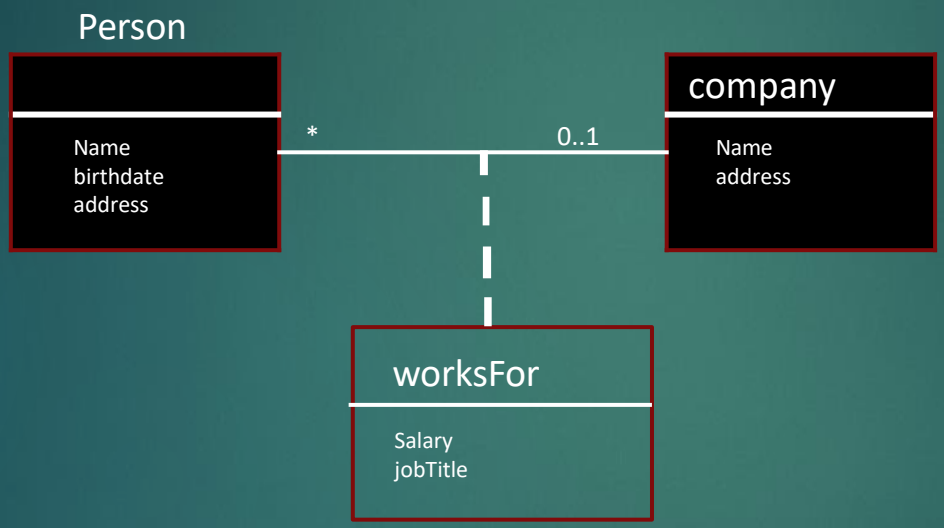
Bags and Sequence

- Multiple links for a pair of objects can be formed at association end with {*bag*} or {*sequence*}
- A bag is a collection of elements with duplicates allowed
- A sequence is an ordered collection of elements with duplicates allowed.
- {Bag} and {sequence} are permitted only for binary associations.
- {ordered} and {sequence} annotations are the same, except that the first disallows duplicates and the other allows them

Association Classes

- An Association class is an association that is also a class
- Association class can have attributes and operations and participate in associations
- Name for association class can be an adverb.
- The UML notation for an association class is a box attached to the association by a dashed line.



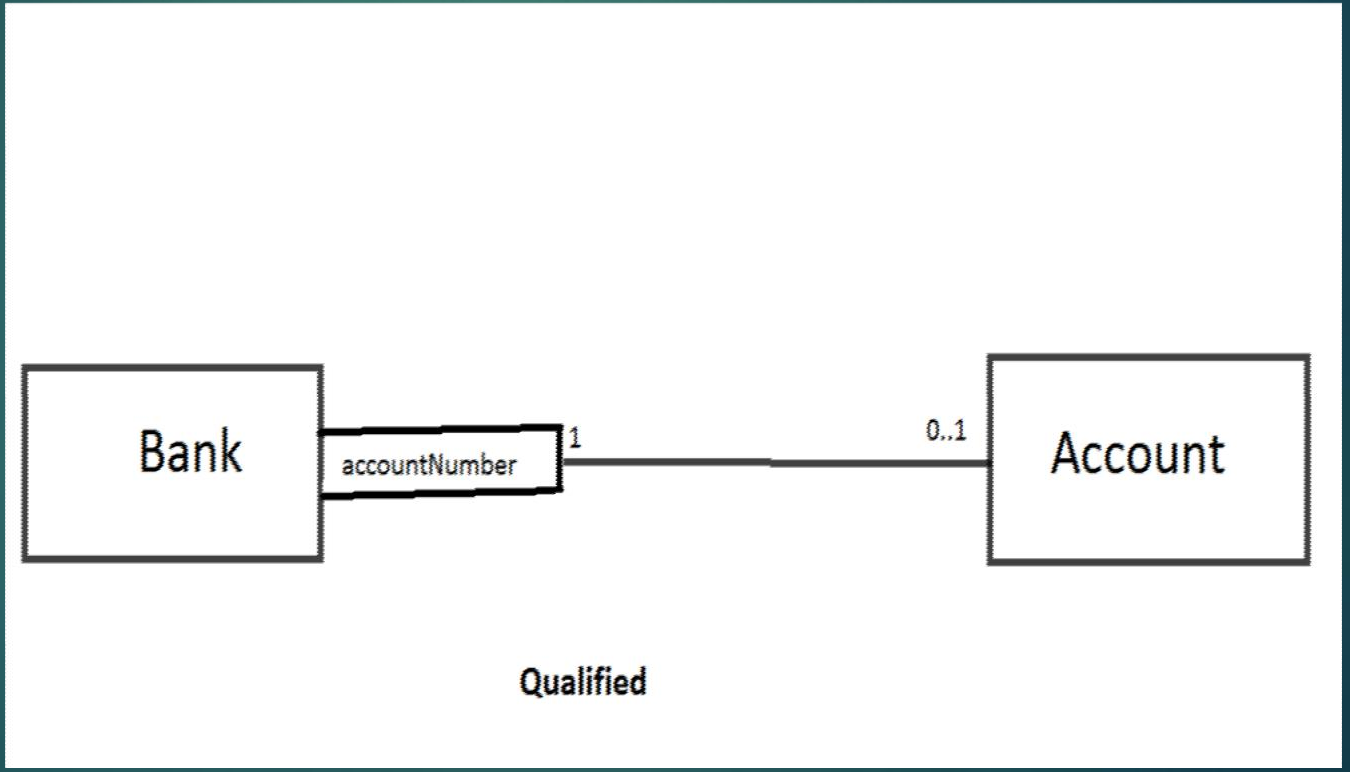

Person
Name
birthdate
address
*
0..1
company
Name
address
worksFor
Salary
jobTitle

Qualified Associations

- It is an association in which an attribute called the qualifier disambiguates the objects for a “many” association end
- It is possible to define qualifiers for one-to-many and many-to-many associations
- A qualifier selects among the target objects , reducing the effective multiplicity from “many” to “one”
- The notation for a qualifier is a small box on the end of the association line near the source class
- The source class plus the the qualifier yields the target class

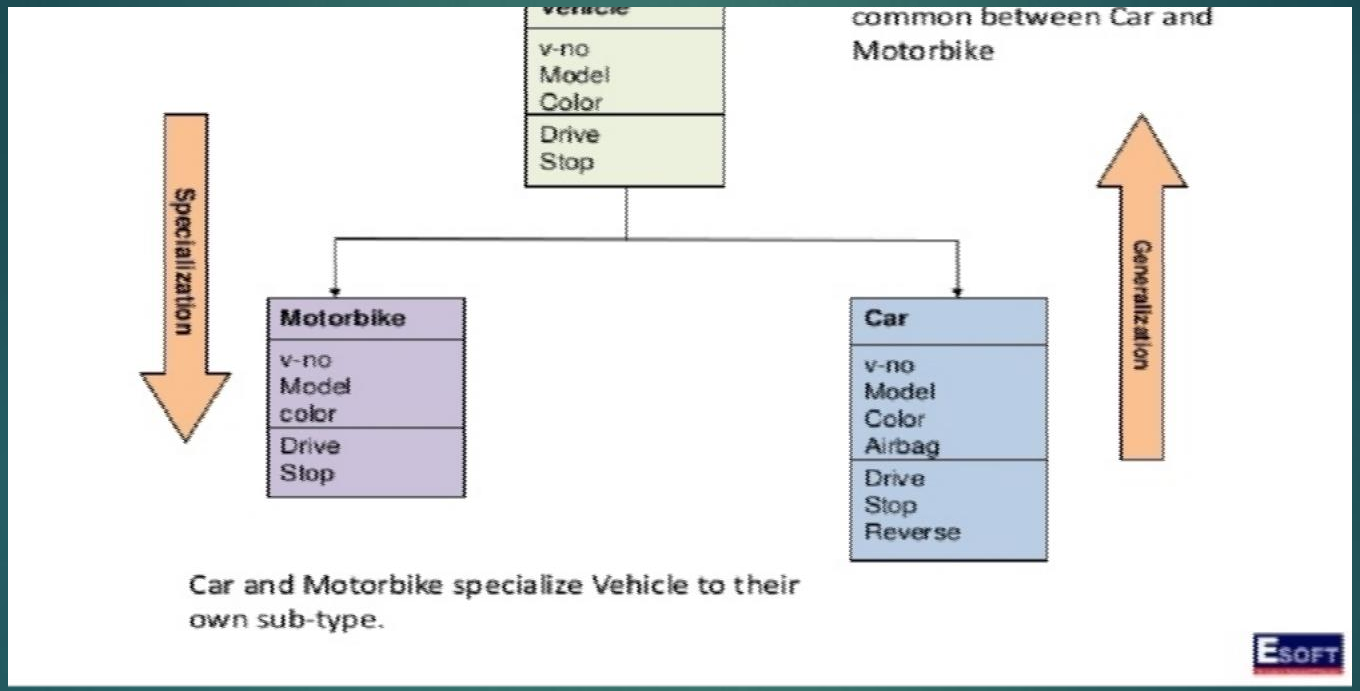
Example: Bank services multiple accounts, an account belongs to a single bank. Within the context of bank , the account number specifies the unique account. Bank and Account are classes, and accountNumber is the qualifier

Not Qualified

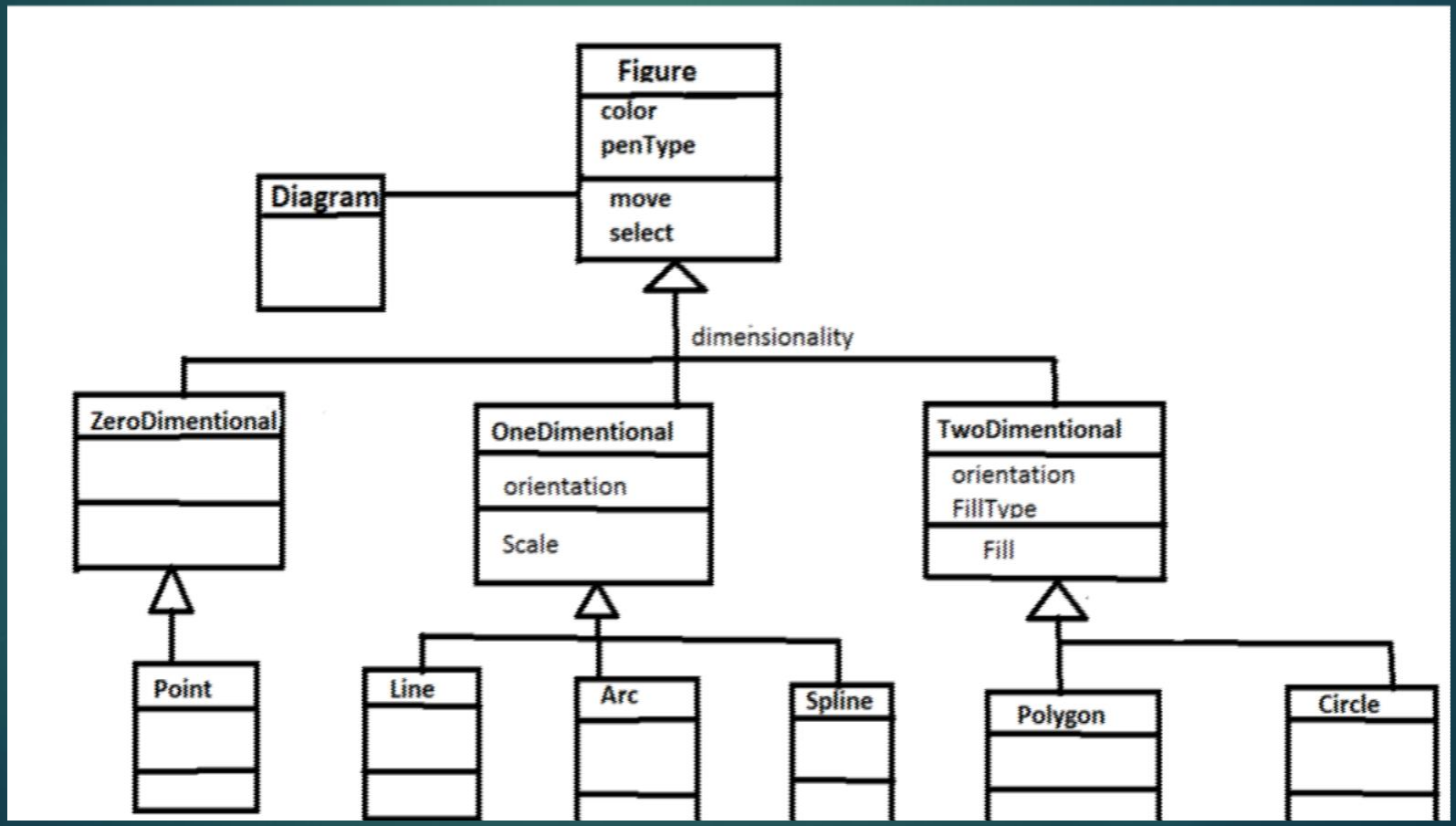

Bank
accountNumber
1
0..1
Account
Qualified

Generalization and Inheritance

- Generalization is the relationship between a class (the superclass) and one or more variations of the class (the subclass)
- Generalization organizes classes by their similarities and differences, structuring the descriptions of the objects
- The superclass holds common attributes, operations and associations
- The subclasses add specific attributes, operations and associations
- Each subclass is said to inherit the features of its superclass
- Generalization is sometimes called the “is-a” relationship, because each instance of a subclass is an instance of the superclass as well.


common between Car and Motorbike
v-no
Model
Color
Drive
Stop
Specialization
Generalization
Motorbike
v-no
Model
color
Drive
Stop
Car
v-no
Model
Color
Airbag
Drive
Stop
Reverse
Car and Motorbike specialize Vehicle to their own sub-type.
ESOFT

Following example shows classes of geometric figures. Here move and select are operations that all subclasses inherit. Where as Scale applies to OneDimensional figures and Fill applies to TwoDimensional Figures.



Use of generalization

Generalization has three purposes

- Support for polymorphism
- Structure the description of objects
- Enable reuse of code

Overriding Features

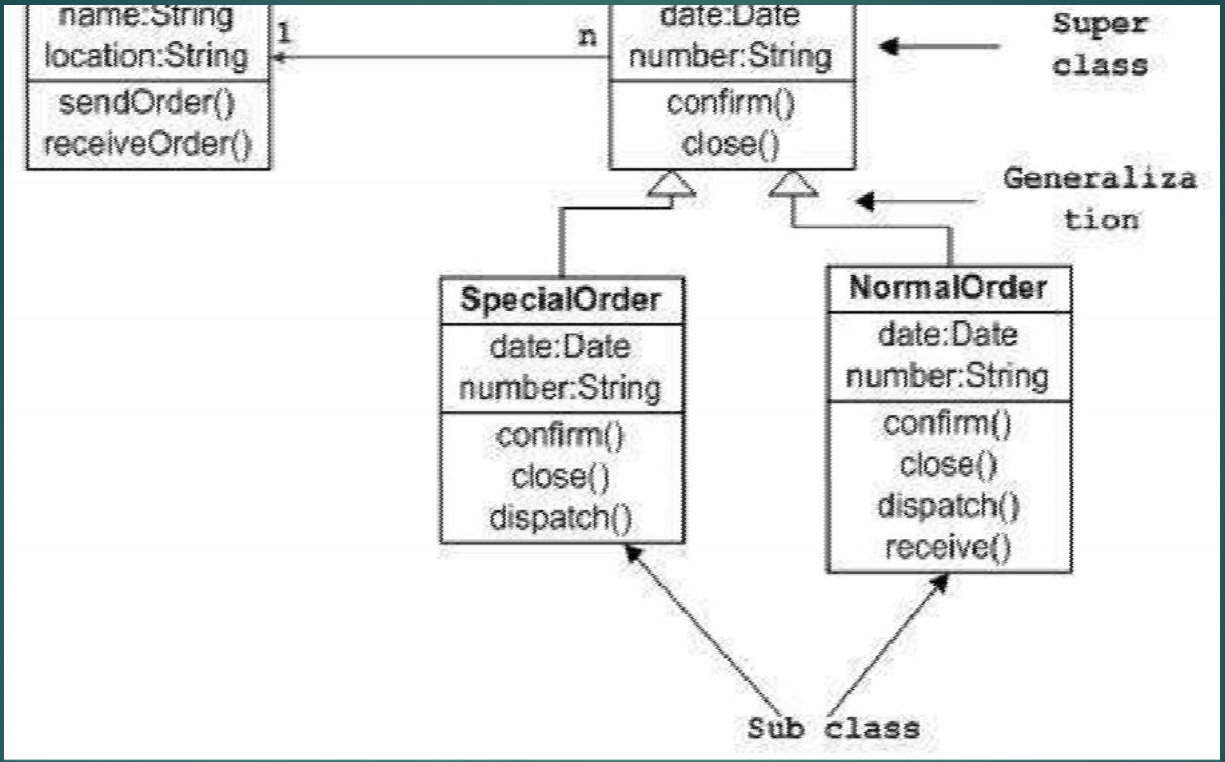
- The subclass may override a superclass feature by defining a feature with the same name.
- Reasons to override feature

  -To specify behavior that depends on the subclass

  -Tighten the specification of the feature

  -Improve performance


Public Attribute
MyClassName
+attribute : int
Private Attribute


name:String
location:String
sendOrder()
receiveOrder()
1
n
date:Date
number:String
confirm()
close()
Super class
Generalization
SpecialOrder
date:Date
number:String
confirm()
close()
dispatch()
NormalOrder
date:Date
number:String
confirm()
close()
dispatch()
receive()
Sub class

UNIT 1: CHAPTER 4 ADVANCED CLASS MODELLING

St Aloysius College (Autonomous) Mangaluru

CONTENTS

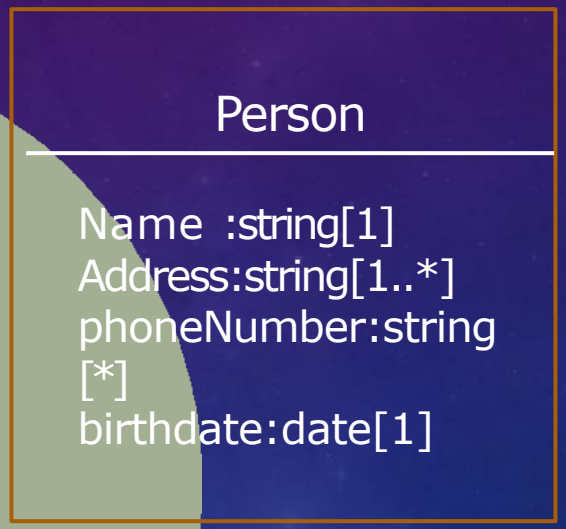
- **Advanced Class Modeling**

Multiplicity, Association Ends, Aggregation, Aggregation versus Association, Aggregation versus Composition

MULTIPLICITY

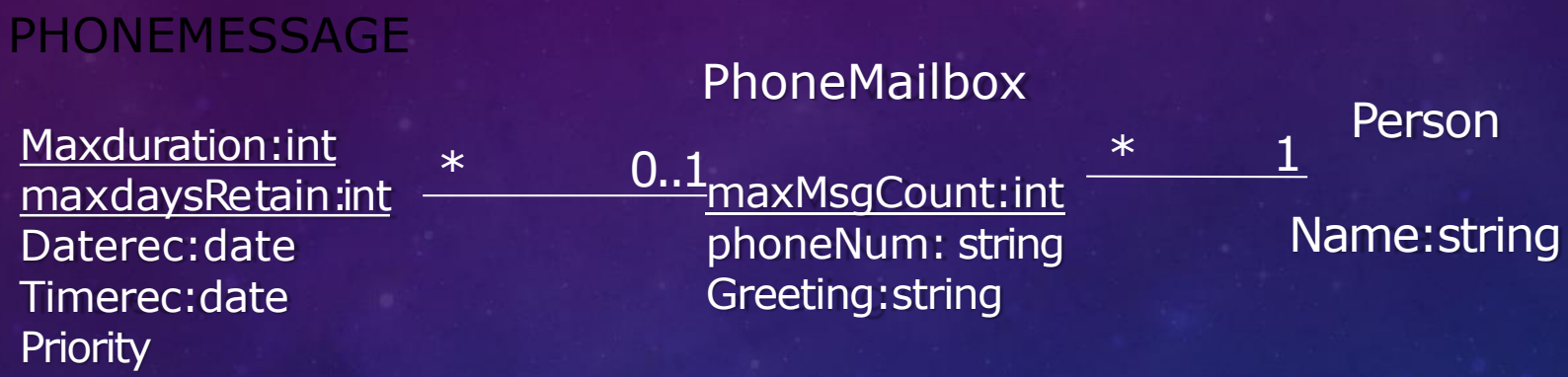
- Multiplicity is a constraint on the cardinality of a set
- Multiplicity is also applies to attributes
- Multiplicity for an attribute specifies the number of possible values for each instantiation of an attribute
- The specifications are [1], [0..1], [*]
- Multiplicity specifies whether an attribute is mandatory or optional
- Multiplicity also indicates if an attribute is single valued or can be a collection

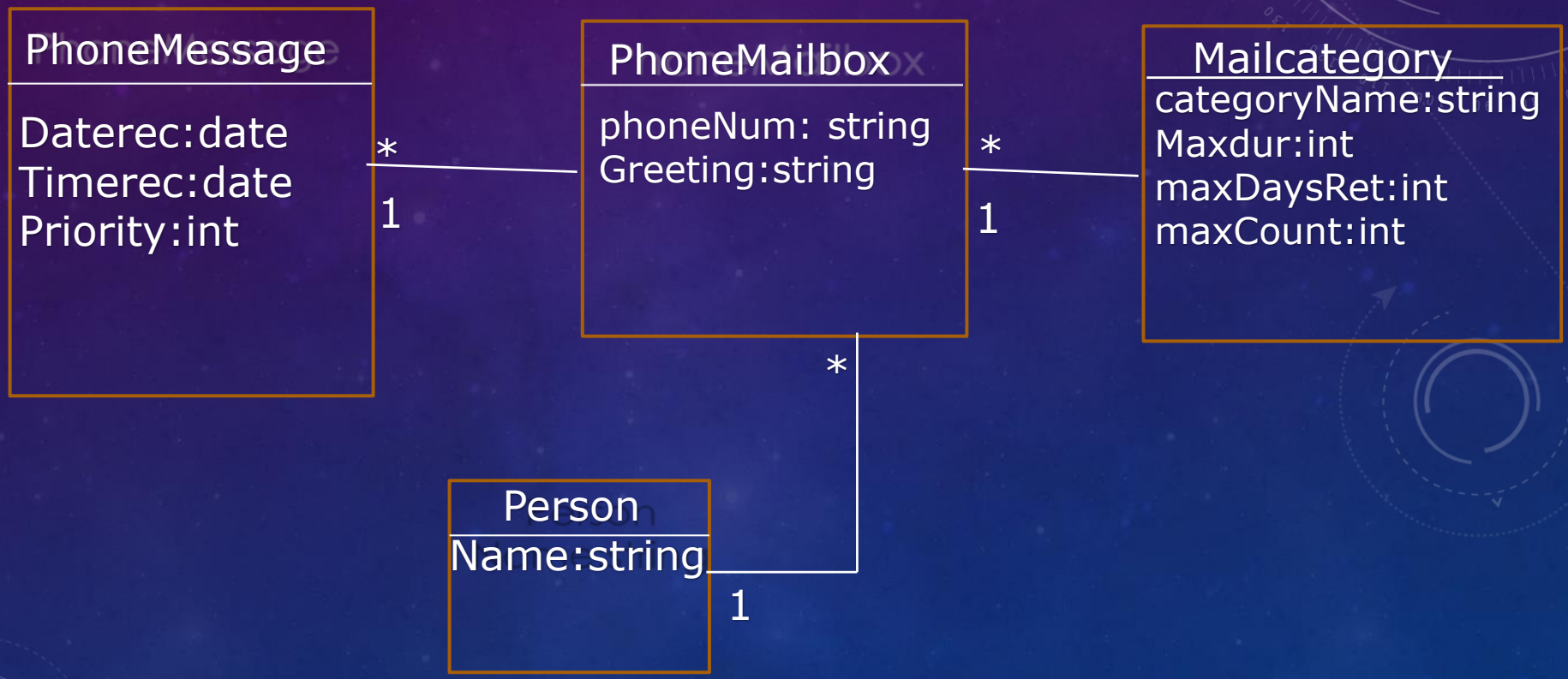
- EXAMPLE


Person
Name :string[1]
Address:string[1..*]
phoneNumber:string
[*]
birthdate:date[1]

SCOPE

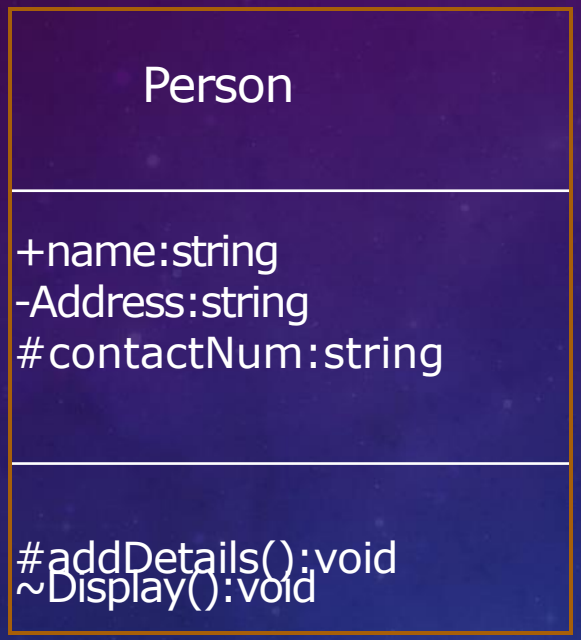
- The scope indicates if a feature applies to an object or a class
- An underline distinguishes features with class scope from those with object scope


PHONEMESSAGE
Maxduration:int
maxdaysRetain:int
Daterec:date
Timerec:date
Priority
*
0..1
PhoneMailbox
maxMsgCount:int
phoneNum: string
Greeting:string
*
1
Person
Name:string


PhoneMessage
Daterec:date
Timerec:date
Priority:int
*
1
PhoneMailbox
phoneNum: string
Greeting:string
*
1
Mailcategory
categoryName:string
Maxdur:int
maxDaysRet:int
maxCount:int
*
Person
Name:string
1

VISIBILITY

- Visibility refers to the ability of a method to reference a feature from another class
- Possible values are
  - public, private, protected, package
    - Any method can access public feature
    - Only methods of containing class can access private feature
    - Only methods of containing class and its descendant class via inheritance can access protected feature
    - Methods of class defined in the same package can access package feature
- UML notations are +,-,#,~


Person
+name:string
-Address:string
#contactNum:string
#addDetails():void
~Display():void

4.2 Association Ends

As the name implies, an ***association end*** is an end of an association. A binary association has two ends, a ternary association (Section 4.3) has three ends, and so forth. Chapter 3 discussed the following properties.

- **Association end name**. An association end may have a name. The names disambiguate multiple references to a class and facilitate navigation. Meaningful names often arise, and it is useful to place the names within the proper context.
- **Multiplicity**. You can specify multiplicity for each association end. The most common multiplicities are "1" (exactly one), "0..1" (at most one), and "*" ("many"—zero or more).
- **Ordering**. The objects for a "many" association end are usually just a set. However, sometimes the objects have an explicit order.
- **Bags and sequences**. The objects for a "many" association end can also be a bag or sequence.
- **Qualification**. One or more qualifier attributes can disambiguate the objects for a "many" association end.

Association ends have some additional properties.

- **Aggregation**. The association end may be an aggregate or constituent part (Section 4.4). Only a binary association can be an aggregation; one association end must be an aggregate and the other must be a constituent.
- **Changeability**. This property specifies the update status of an association end. The possibilities are *changeable* (can be updated) and *readonly* (can only be initialized).
- **Navigability**. Conceptually, an association may be traversed in either direction. However, an implementation may support only one direction. The UML shows navigability with an arrowhead on the association end attached to the target class. Arrowheads may be attached to zero, one, or both ends of an association.
- **Visibility**. Similar to attributes and operations (Section 4.1.4), association ends may be *public*, *protected*, *private*, or *package*.

4.3 N-ary Associations

Chapter 3 presented binary associations (associations between two classes). However, you may occasionally encounter ***n-ary associations*** (associations among three or more classes.) You should try to avoid n-ary associations—most of them can be decomposed into binary associations, with possible qualifiers and attributes. Figure 4.5 shows an association that at first glance might seem to be an n-ary but can readily be restated as binary associations.

A nonatomic n-ary association—a person makes the purchase of stock in a company...

Can be restated as...

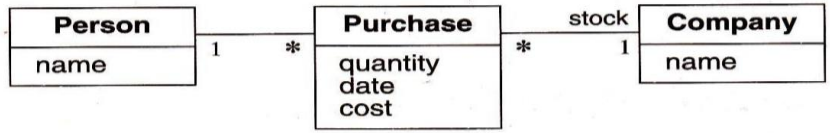
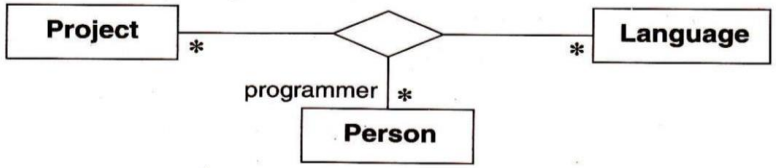


Figure 4.5 Restating an n-ary association. You can decompose most n-ary associations into binary associations.

Figure 4.6 shows a genuine n-ary (ternary) association: Programmers use computer languages on projects. This n-ary association is an atomic unit and cannot be subdivided into binary associations without losing information. A programmer may know a language and work on a project, but might not use the language on the project. The UML symbol for n-ary associations is a diamond with lines connecting to related classes. If the association has a name, it is written in italics next to the diamond.

Class diagram



Instance diagram

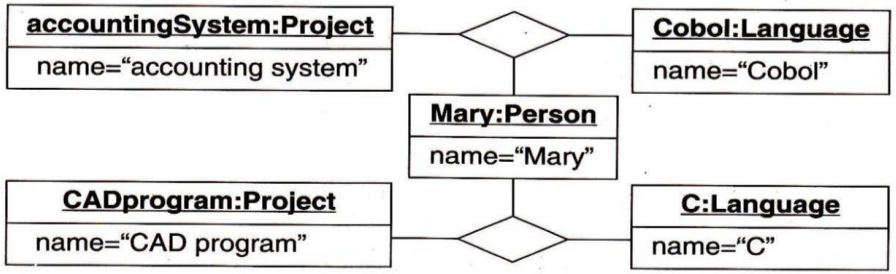


Figure 4.6 Ternary association and links. An n-ary association can have association end names, just like a binary association.

As Figure 4.6 illustrates, an n-ary association can have a name for each end just like a binary association. End names are necessary if a class participates in an n-ary association more than once. You cannot traverse n-ary associations from one end to another as with binary associations, so end names do not represent pseudo attributes of the participating classes. The OCL [Warmer-99] does not define notation for traversing n-ary associations.

Figure 4.7 shows another ternary association: A professor teaches a listed course during a semester. The resulting delivered course has a room number and any number of textbooks.

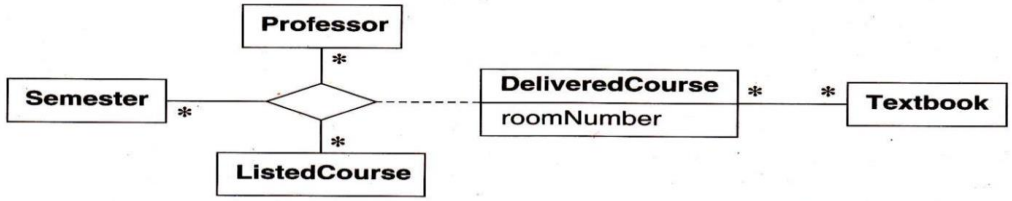


Figure 4.7 Another ternary association. N-ary associations are full-fledged associations and can have association classes.

The typical programming language cannot express n-ary associations. Thus if you are programming, you will need to promote n-ary associations to classes as Figure 4.8 does for *DeliveredClass*. Be aware that you change the meaning of a model, when you promote an n-ary association to a class. An n-ary association enforces that there is at most one link for each combination—for each combination of *Professor*, *Semester*, and *ListedCourse* in Figure 4.7 there is one *DeliveredCourse*. In contrast a promoted class permits any number of links—for each combination of *Professor*, *Semester*, and *ListedCourse* in Figure 4.8 there can be many *DeliveredCourses*. If you were implementing Figure 4.8, special application code would have to enforce the uniqueness of *Professor* + *Semester* + *ListedCourse*.

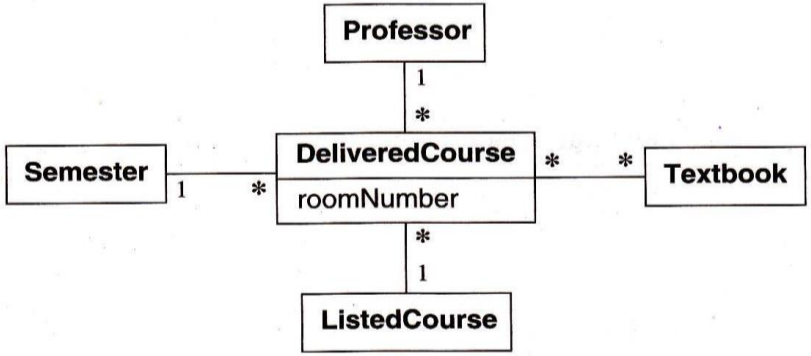


Figure 4.8 Promoting an n-ary association. Programming languages cannot express n-ary associations, so you must promote them to classes.

4.4 Aggregation

Aggregation is a strong form of association in which an aggregate object is *made of* constituent parts. Constituents are *part of* the aggregate. The aggregate is semantically an extended object that is treated as a unit in many operations, although physically it is made of several lesser objects.

We define an aggregation as relating an assembly class to *one* constituent part class. An assembly with many kinds of constituent parts corresponds to many aggregations. For example, a *LawnMower* consists of a *Blade*, an *Engine*, many *Wheels*, and a *Deck*. *LawnMower* is the assembly and the other parts are constituents. *LawnMower* to *Blade* is one aggregation, *LawnMower* to *Engine* is another aggregation, and so on. We define each individual pairing as an aggregation so that we can specify the multiplicity of each constituent part within the assembly. This definition emphasizes that aggregation is a special form of binary association.

The most significant property of aggregation is ***transitivity***—that is, if *A* is part of *B* and *B* is part of *C*, then *A* is part of *C*. Aggregation is also ***antisymmetric***—that is, if *A* is part of *B*, then *B* is not part of *A*. Many aggregate operations imply transitive closure* and operate on both direct and indirect parts.

4.4.1 Aggregation Versus Association

Aggregation is a special form of association, not an independent concept. Aggregation adds semantic connotations. If two objects are tightly bound by a part-whole relationship, it is an aggregation. If the two objects are usually considered as independent, even though they may often be linked, it is an association. Some tests include:

- Would you use the phrase *part of*?
- Do some operations on the whole automatically apply to its parts?
- Do some attribute values propagate from the whole to all or some parts?
- Is there an intrinsic asymmetry to the association, where one class is subordinate to the other?

Aggregations include bill-of-materials, part explosions, and expansions of an object into constituent parts. Aggregation is drawn like association, except a small diamond indicates the assembly end. In Figure 4.9 a lawn mower consists of one blade, one engine, many wheels, and one deck. The manufacturing process is flexible and largely combines standard parts, so blades, engines, wheels, and decks pertain to multiple lawn mower designs.

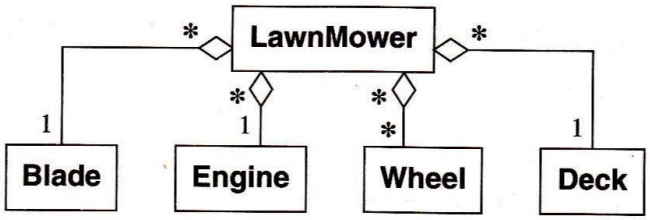


Figure 4.9 **Aggregation**. Aggregation is a kind of association in which an aggregate object is made of constituent parts.

The decision to use aggregation is a matter of judgment and can be arbitrary. Often it is not obvious if an association should be modeled as an aggregation. To a large extent this kind of uncertainty is typical of modeling; modeling requires seasoned judgment and there are few hard and fast rules. Our experience has been that if you exercise careful judgment and are consistent, the imprecise distinction between aggregation and ordinary association does not cause problems in practice.

4.4.2 Aggregation Versus Composition

The UML has two forms of part-whole relationships: a general form called *aggregation* and a more restrictive form called *composition*.

Composition is a form of aggregation with two additional constraints. A constituent part can belong to at most one assembly. Furthermore, once a constituent part has been assigned an assembly, it has a coincident lifetime with the assembly. Thus composition implies ownership of the parts by the whole. This can be convenient for programming: Deletion of an assembly object triggers deletion of all constituent objects via composition. The notation for composition is a small *solid* diamond next to the assembly class (vs. a small *hollow* diamond for the general form of aggregation).

In Figure 4.10 a company consists of divisions, which in turn consist of departments; a company is indirectly a composition of departments. A company is not a composition of its employees, since company and person are independent objects of equal stature.

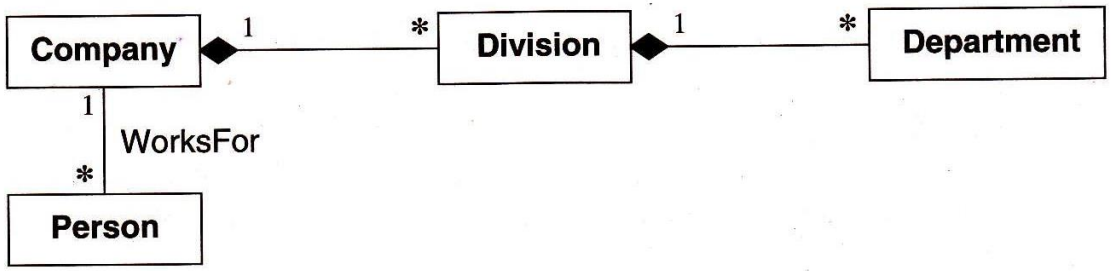


Figure 4.10 Composition. With composition a constituent part belongs to at most one assembly and has a coincident lifetime with the assembly.