OBJECT ORIENTED MODELING AND DESING PATTERNS

MCA DEPT, BIET **SYLLABUS 1st and 2nd Chapter**
1. Introduction, Modeling Concepts: What is Object Orientation? What is OO development? OO themes; Evidence for usefulness of OO development; OO modeling history. Modeling as Design Technique: Modeling; abstraction; The three models
2. Class Modeling and Advanced Class Modeling: Object and class concepts; Link and associations concepts; Generalization and inheritance; A sample class model; Navigation of class models; Practical tips. Advanced object and class concepts; Association ends; N-array associations; Aggregation; Abstract classes; Multiple inheritance; Metadata; Reification; Constraints; Derived data; Packages; Practical tips

TEXT BOOKS:

Text Books:

1. Michael Blaha, James Rumbaugh: Object-Oriented Modeling and Design with UML, 2nd Edition, Pearson Education / PHI, 2005. (Chapters 1 to 15)
2. Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal: Pattern-Oriented Software Architecture, A System of Patterns, Volume 1, John Wiley and Sons, 2006. (Chapters 1, 3)

Reference Books:

1. Grady Booch et al: Object-Oriented Analysis and Design with Applications, 3rd Edition, Pearson, 2007.
2. Mark Priestley: Practical Object-Oriented Design with UML, 2nd Edition, Tata McGraw-Hill, 2003.
3. K. Barclay, J. Savage: Object-Oriented Design with UML and JAVA, Elsevier, 2008.
4. Booch, G., Rumbaugh, J., and Jacobson, I.: The Unified Modeling Language User Guide, 2nd Edition, Pearson, 2005.
5. E. Gamma, R. Helm, R. Johnson, J. Vlissides: Design Patterns-Elements of Reusable Object- Oriented Software, Addison-Wesley, 1995.
6. Simon Bennett, Steve McRobb and Ray Farmer: Object-Oriented Systems Analysis and Design Using UML, 2nd Edition, Tata McGraw-Hill, 2002.
7. Rumbaugh, Blaha, Premerhani, Eddy, Lorensen; Object Oriented Modeling and Design, PHI Latest Edition

Unit1: Introduction, Modeling Concepts:

INTRODUCTION

Note 1:

Intention of this subject (object oriented modeling and design) is to learn how to apply object -oriented concepts to all the stages of the software development life cycle.

Note 2:

Object-oriented modeling and design is a way of thinking about problems using models organized around real world concepts. The fundamental construct is the object, which combines both data structure and behavior.

WHAT IS OBJECT ORIENTATION?

_**Definition:** OO means that we organize software as a collection of discrete objects (that incorporate both data structure and behavior).

_There are four**aspects (characteristics)** required by an OO approacho

Identity.

- Classification.
- Inheritance.
- Polymorphism.

_**Identity:**

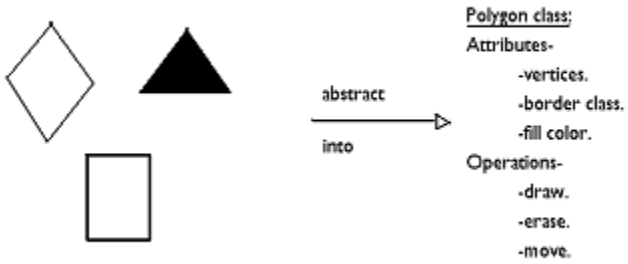
- **Identity** means that data is quantized into discrete, distinguishable entities called objects.
- **E.g. for objects:** personal computer, bicycle, queen in chess etc.

- Objects can be concrete (such as a file in a file system) or conceptual (such as scheduling policy in a multiprocessing OS). Each object has its own inherent identity. (i.e two objects are distinct even if all their attribute values are identical).
- In programming languages, an object is referenced by a unique handle.

_Classification:

- **Classification** means that objects with the same data structure (attribute) and behavior (operations) are grouped into a class.
- E.g. paragraph, monitor, chess piece.
- Each object is said to be an instance of its class.
- Fig below shows objects and classes: Each class describes a possibly infinite set of individual objects.

Eg:



Bycycle class:
Attributes-
- frame size
-wheel size
no. of gears
-material
Operations-
-shift
-move
-repair

Inheritance:

- It is the sharing of attributes and operations (features) among classes based on a hierarchical relationship. A super class has general information that sub classes refine and elaborate.
- E.g. Scrolling window and fixed window are sub classes of window.

_Polymorphism:

- **Polymorphism** means that the same operation may behave differently for different classes.
- For E.g. move operation behaves differently for a pawn than for the queen in a chess game.

Note: An *operation* is a procedure/transformation that an object performs or is subjected to. An implementation of an operation by a specific class is called a *method*.

WHAT IS OO DEVELOPMENT?

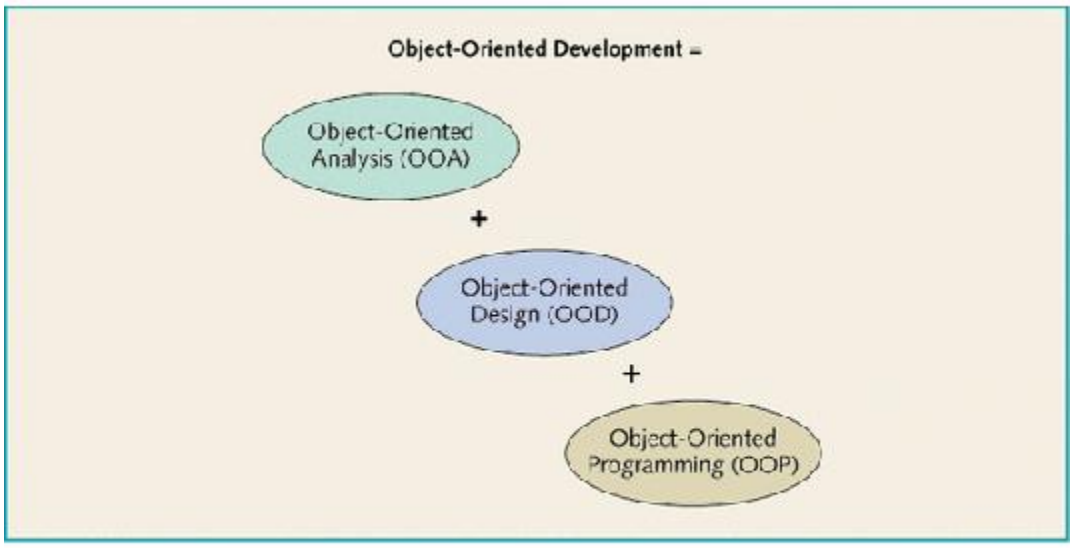


Figure 1-3 Object-oriented development

_**Development** refers to the software life cycle: Analysis, Design and Implementation. The essence of OO Development is the *identification* and *organization* of application concepts, rather than their final representation in a programming language. It's a conceptual process independent of programming languages. OO development is fundamentally a way of thinking and not a programming technique.

OO methodology

_Here we present a process for OO development and a graphical notation for representing OO concepts. The process consists of building a model of an application and then adding details to it during design.

_The methodology has the following stages

- **System conception:** Software development begins with business analysis or users conceiving an application and formulating tentative requirements.
- **Analysis:** The analyst scrutinizes and rigorously restates the requirements from the system conception by constructing models. The analysis model is a concise, precise abstraction of what the desired system must do, not how it will be done.
- The analysis model has two parts-

- _ **Domain Model**- a description of real world objects reflected within the system.
- _ **Application Model**- a description of parts of the application system itself that are visible to the user.
- E.g. In case of stock broker application-
- Domain objects may include- stock, bond, trade & commission.
- Application objects might control the execution of trades and present the results.
- **System Design:** The development teams devise a high-level strategy- The System Architecture- for solving the application problem. The system designer should decide what performance characteristics to optimize, chose a strategy of attacking the problem, and make tentative resource allocations.
- **Class Design:** The class designer adds details to the analysis model in accordance with the system design strategy. His focus is the data structures and algorithms needed to implement each class.
- **Implementation:** Implementers translate the classes and relationships developed during class design into a particular programming language, database or hardware. During implementation, it is important to follow good software engineering practice.

Three models

_ We use three kinds of models to describe a system from different view points.

1. **Class Model—**for the objects in the system & their relationships.

It describes the static structure of the objects in the system and their relationships.

Class model contains class diagrams- a graph whose nodes are classes and arcs are relationships among the classes.

2. **State model—**for the life history of objects.

It describes the aspects of an object that change over time. It specifies and implements control with state diagrams-a graph whose nodes are states and whose arcs are transition between states caused by events.

3. **Interaction Model**—for the interaction among objects.

It describes how the objects in the system co-operate to achieve broader results. This model starts with use cases that are then elaborated with sequence and activity diagrams.

Use case – focuses on functionality of a system – i.e what a system does for users.

Sequence diagrams – shows the object that interact and the time sequence of their interactions.

Activity diagrams – elaborates important processing steps.

OO THEMES

Several themes pervade OO technology. Few are –

1. **Abstraction**

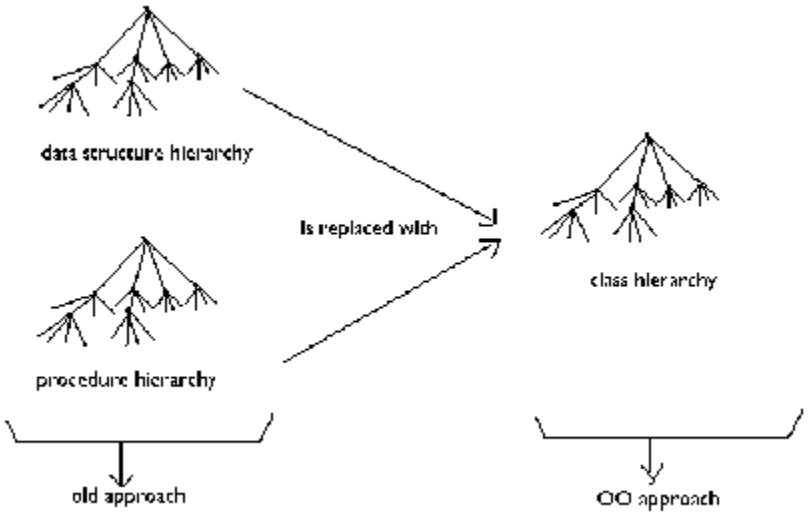
- Abstraction lets you focus on essential aspects of an application while ignoring details i.e focusing on what an object is and does, before deciding how to implement it.
- It's the most important skill required for OO development.

2. **Encapsulation (information hiding)**

- It 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 a small change has massive ripple effects.

3. **Combining data and behavior**

- Caller of an operation need not consider how many implementations exist.
- In OO system the data structure hierarchy matches the operation inheritance
- hierarchy (fig).



4. Sharing

- OO techniques provide sharing at different levels.
- Inheritance of both data structure and behavior lets sub classes share common code.
- OO development not only lets you share information within an application, but also offers the prospect of reusing designs and code on future projects.

5. **Emphasis on the essence of an object**

- OO development places a greater emphasis on data structure and a lesser emphasis on procedure structure than functional-decomposition methodologies.

6. **Synergy**

- Identity, classification, polymorphism and inheritance characterize OO languages.

- Each of these concepts can be used in isolation, but together they complement each other synergistically.

MODELLING AS A DESIGN TECHNIQUE

Note: A model is an abstraction of something for the purpose of understanding it before building it.

MODELLING

_ Designers build many kinds of models for various purposes before constructing things.

_ Models serve several purposes–

- **Testing a physical entity before building it:** Medieval built scale models of Gothic Cathedrals to test the forces on the structures. Engineers test scale models of airplanes, cars and boats to improve their dynamics.
- **Communication with customers:** Architects and product designers build models to show their customers (note: mock-ups are demonstration products that imitate some of the external behavior of a system).
- **Visualization:** Storyboards of movies, TV shows and advertisements let writers see how their ideas flow.
- **Reduction of complexity:** Models reduce complexity to understand directly by separating out a small number of important things to do with at a time.

ABSTRACTION

_**Abstraction** is the selective examination of certain aspects of a problem.

_ The goal of abstraction is to isolate those aspects that are important for some purpose and suppress those aspects that are unimportant.

OO modeling history (page number : 09)

THE THREE MODELS

1. **Class Model**: represents the static, structural, "data" aspects of a system.

- It describes the structure of objects in a system- their identity, their relationships to other objects, their attributes, and their operations.
- Goal in constructing class model is to capture those concepts from the real world that are important to an application.
- Class diagrams express the class model.

2. **State Model**: represents the temporal, behavioral, "control" aspects of a system.

- State model describes those aspects of objects concerned with time and the sequencing of operations – events that mark changes, states that define the context for events, and the organization of events and states.
- State diagram express the state model.
- Each state diagram shows the state and event sequences permitted in a system for one class of objects.

State diagram refer to the other models.

- Actions and events in a state diagram become operations on objects in the class model. References between state diagrams become interactions in the interaction model.

3. **Interaction model** – represents the collaboration of individual objects, the "interaction" aspects of a system.

- Interaction model describes interactions between objects – how individual objects collaborate to achieve the behavior of the system as a whole.
- The state and interaction models describe different aspects of behavior, and you need both to describe behavior fully.
- Use cases, sequence diagrams and activity diagrams document the interaction model.

----------oooooooo----------

Unit 2: **Class Modeling and Advanced Class Modeling**

CLASS MODELLING

Note: A class model captures the static structure of a system by characterizing the objects in the system, the relationships between the objects, and the attributes and operations for each class of objects.

OBJECT AND CLASS CONCEPT

Objects

_ Purpose of class modeling is to describe objects.

_ An **object** is a concept, abstraction or thing with identity that has meaning for an application.

Ex: Joe Smith, Infosys Company, process number 7648 and top window are objects.

Classes

_ An object is an instance or occurrence of a class

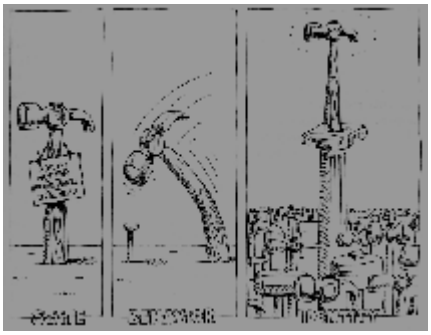
_ A **class** describes a group of objects with the same properties (attributes), behavior (operations), kinds of relationships and semantics.

Ex: Person, company, process and window are classes.

Note: All objects have identity and are distinguishable. Two apples with same color, shape and texture are still individual apples: a person can eat one and then the other. The term identity means that the objects are distinguished by their inherent existence and not by descriptive properties that they may have.

CLASS MODELLING

- **OBJECT AND CLASS CONCEPT**
- An **object** has three characteristics: **state**, **behavior** and **a unique identification**. or
- **An object is a concept, abstraction or thing with identity that has meaning for an application. Eg:**
- Note: The term identity means that the objects are distinguished by their inherent existence and not by descriptive properties that they may have.



Class diagrams

_Class diagrams provide a graphic notation for modeling classes and their relationships, thereby describing possible objects.

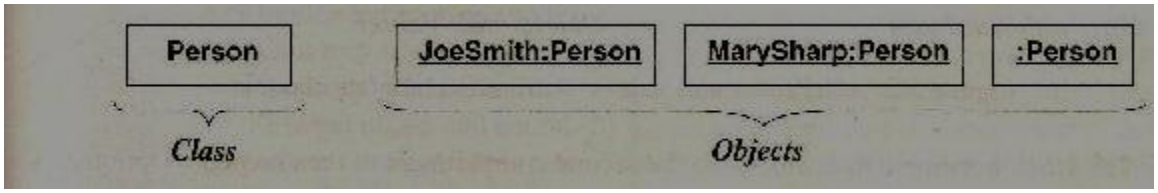
Note: An object diagram shows individual objects and their relationships.

Useful for documenting test cases and discussing examples.

_Class diagrams are useful both for abstract modeling and for designing actual programs.

Note: A class diagram corresponds to infinite set of object diagrams.

_Figure below shows a class (left) and instances (right) described by it.



_Conventions used (UML):

- UML symbol for both classes and objects is box.
- Objects are modeled using box with object name followed by colon followed by class name.
- Use boldface to list class name, center the name in the box and capitalize the first letter. Use singular nouns for names of classes.
- To run together multiword names (such as JoeSmith), separate the words with
- intervening capital letter.

Values and Attributes: _Value

is a piece of data.

Attribute is a named property of a class that describes a value held by each object of the class.

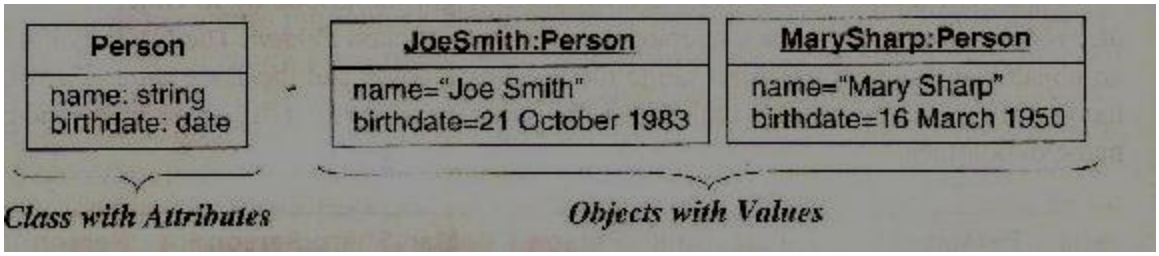
_ Following analogy holds:

Object is to class as value is to attribute.

_ E.g. Attributes: Name, bdate, weight.

Values: JoeSmith, 21 October 1983, 64. (Of person object).

_ Fig shows modeling notation



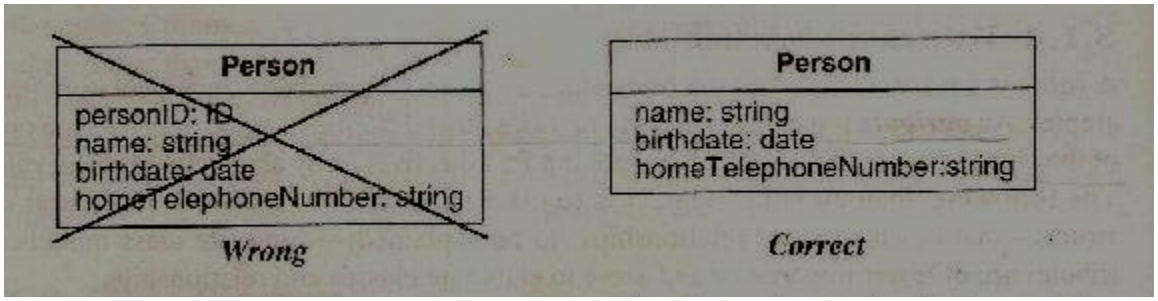
_Conventions used (UML):

- List attributes in the 2nd compartment of the class box. Optional details (like default value) may follow each attribute.
- A colon precedes the type, an equal sign precedes default value.
- Show attribute name in regular face, left align the name in the box and use small case for the first letter.

_ Similarly we may also include attribute values in the 2nd compartment of object boxes with same conventions.

Note: Do not list object identifiers; they are implicit in models.

E.g.



An **operation** is a function or procedure that maybe applied to or by objects in a class.

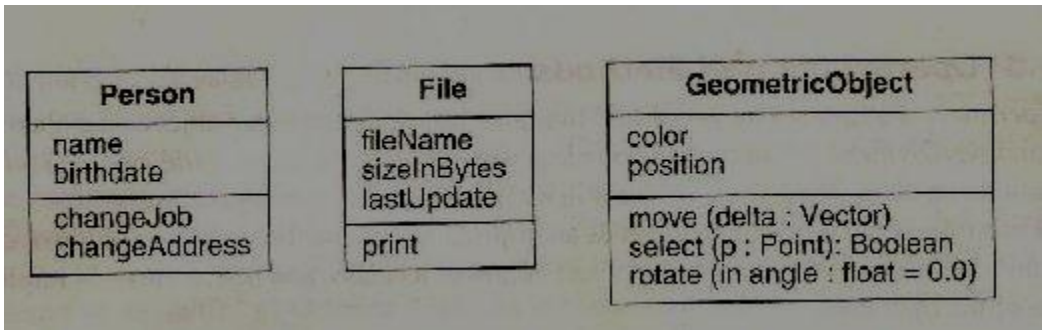
E.g. Hire, fire and pay dividend are operations on Class Company. Open, close, hide and redisplay are operations on class window.

_ A**method** is the implementation of an operation for a class.

E.g. In class file, print is an operation you could implement different methods to print files.

_Note: Same operation may apply to many different classes. Such an operation is polymorphic.

_ Fgi shows modeling notation.



_UML conventions used –

- List operations in 3rd compartment of class box.
- List operation name in regular face, left align and use lower case for first letter.
- Optional details like argument list and return type may follow each operation name.
- Parenthesis enclose an argument list, commas separate the arguments. A colon precedes the result type.

_Note: We do not list operations for objects, because they do not vary among objects of same class.

Summary of Notation for classes

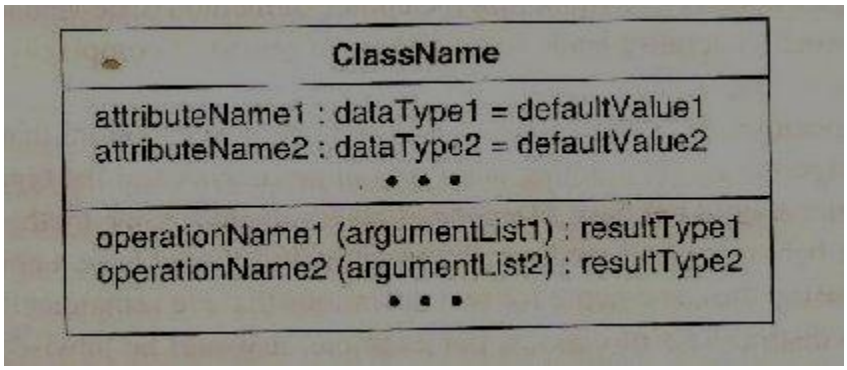
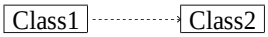
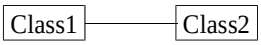


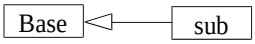
Fig: Summary of modeling notation for classes

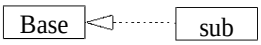
Fig: Notation for an argument of an operation

Class Digarms: Relationships

- Classes can related to each other through different relationships:
  - Dependency 

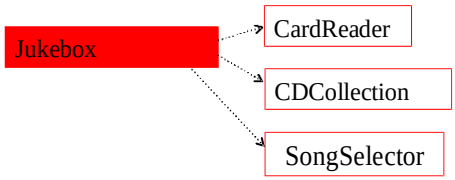
  - Association (delegation) 

  - Generalization (inheritance) 

  - Realization (interfaces) 

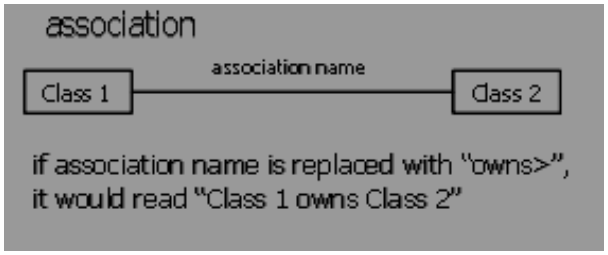
1) Dependency: A Uses Relationship

- Dependencies
  - occurs when one object depends on another
  - if you change one object's interface, you need to change the dependent object
  - arrow points from dependent to needed objects



2)Association: Structural Relationship

- **Association**
  - a relationship between classes indicates some meaningful and interesting connection
  - Can label associations with a hyphen connected verb phrase which reads well between concepts



LINK AND ASSOCIATION CONCEPTS

Note: Links and associations are the means for establishing relationships among objects and classes.

Links and associations

_ A**link** is a physical or conceptual connection among objects.

E.g. JoeSmith *WorksFor* Simplex Company.

_ Mathematically, we define a link as a tuple– that is, a list of objects.

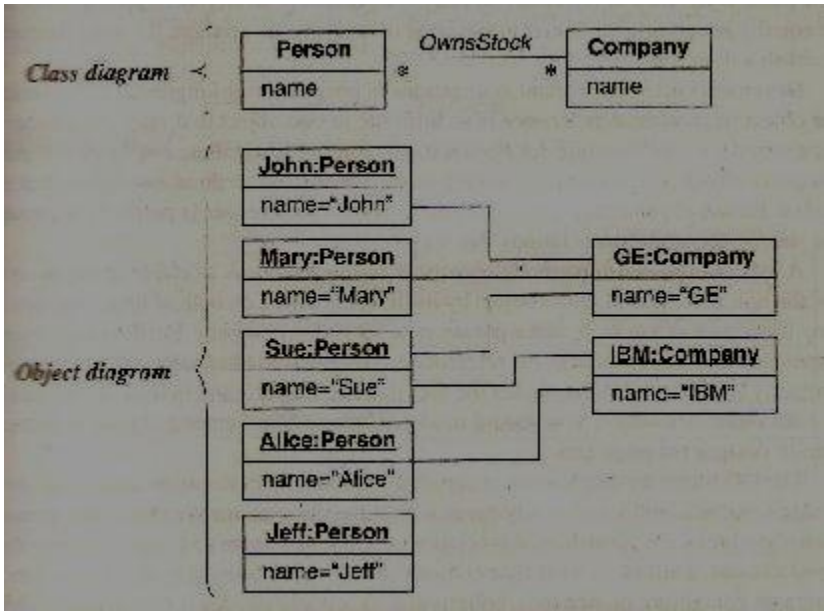
_ A link is an instance of an**association**.

_ An**association** is a description of a group of links with common structure and common semantics.

E.g. a person *WorksFor* a company.

_ An association describes a set of potential links in the same way that a class describes a set of potential objects.

_ Fig shows many-to-many association (model for a financial application).

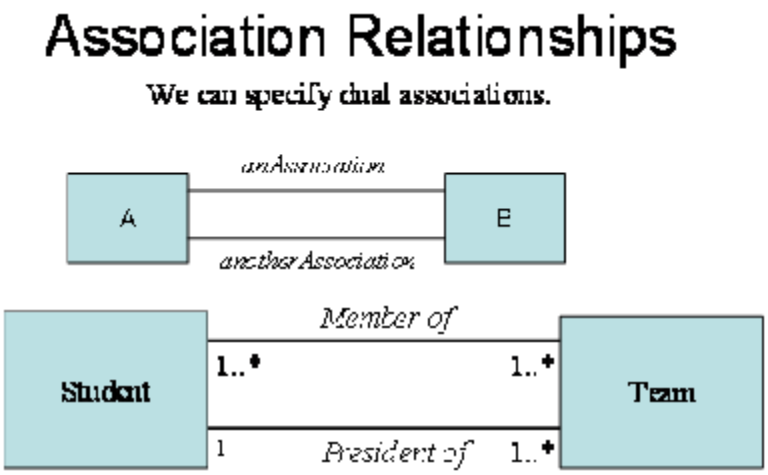


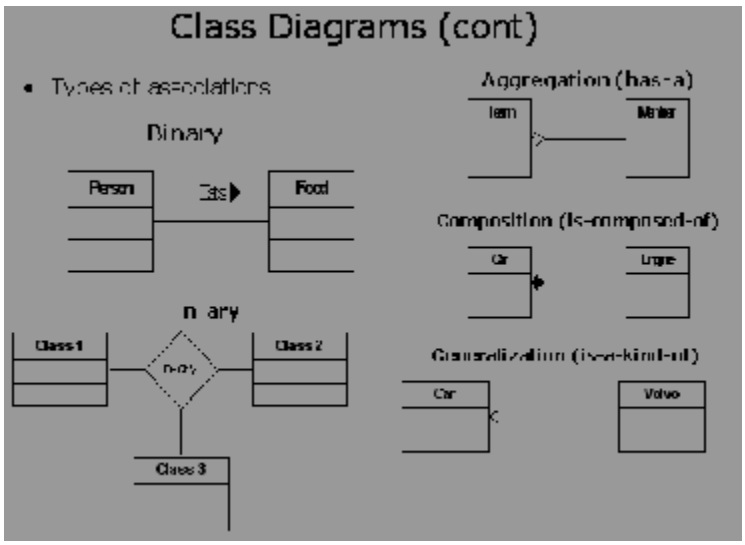
_Conventions used (UML):

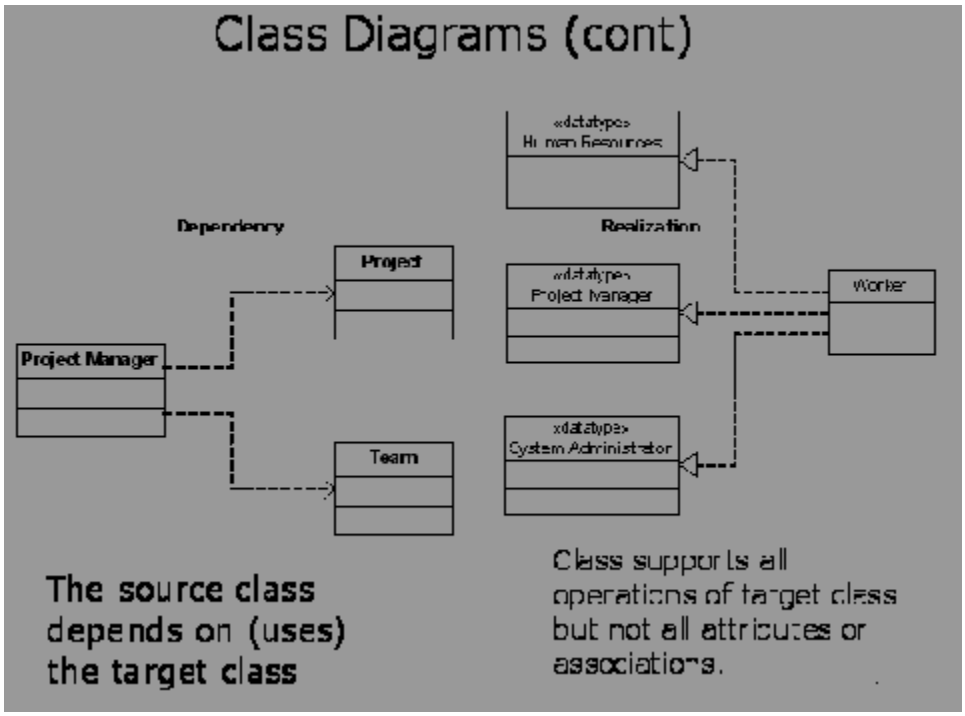
- Link is a line between objects; a line may consist of several line segments.
- If the link has the name, it is underlined.
- Association connects related classes and is also denoted by a line.
- Show link and association names in italics.

_Note:

- Association name is optional, if the model is unambiguous. Ambiguity arises when a model has multiple associations among same classes.
- Developers often implement associations in programming languages as references from one object to another. A reference is an attribute in one object that refers to another object.







Multiplicity

_**Multiplicity** specifies the number of instances of one class that may relate to a single instance of an associated class. Multiplicity constrains the number of related objects.

_**UML conventions:**

- UML diagrams explicitly lists multiplicity at the ends of association lines.
- UML specifies multiplicity with an interval, such as

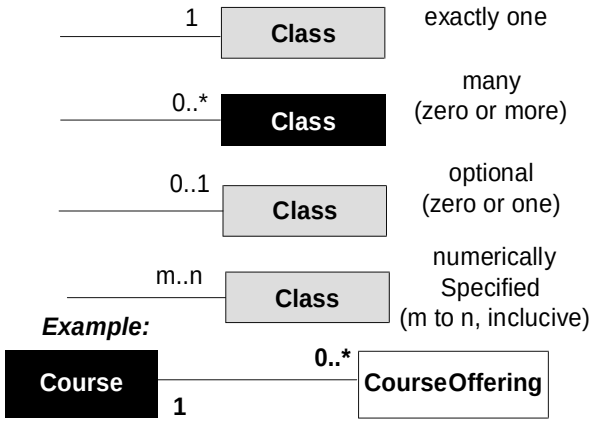
"1" (exactly one).

"1.."(one or more).

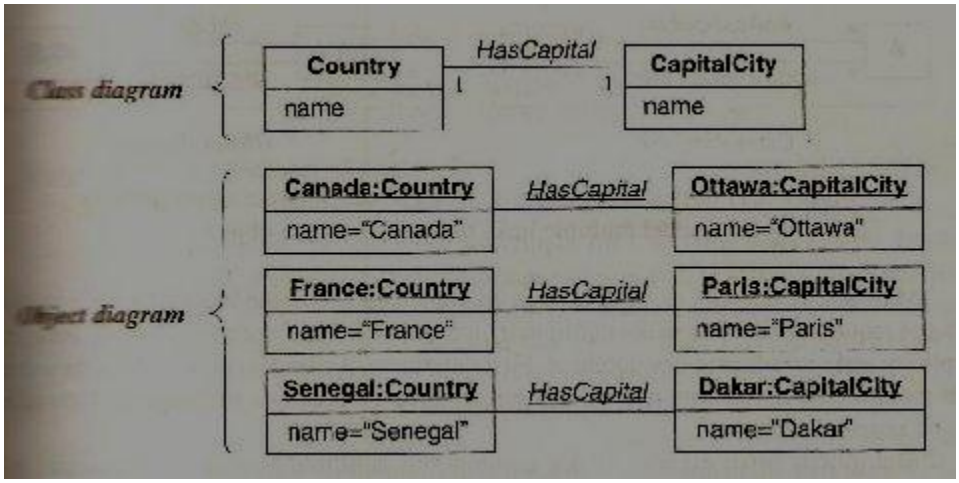
"3..5"(three to five, inclusive).

" * " (many, i.e zero or more).

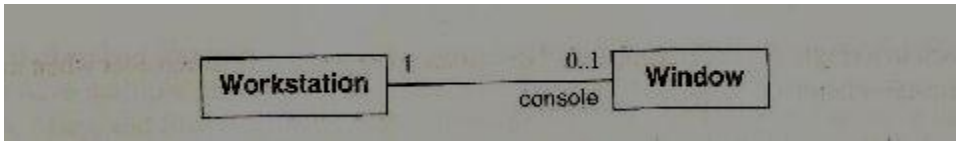
- notations



_ Previous figure illustrates many-to-many multiplicity. Below figure illustrates **one-to-one multiplicity.**



_ Below figure illustrates zero-or-one multiplicity.



_ **Note 1:** Association vs Link.

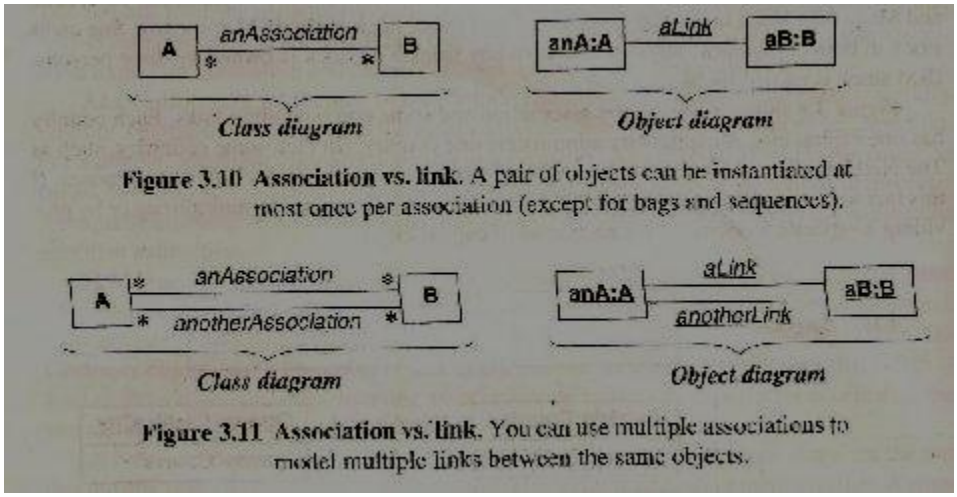


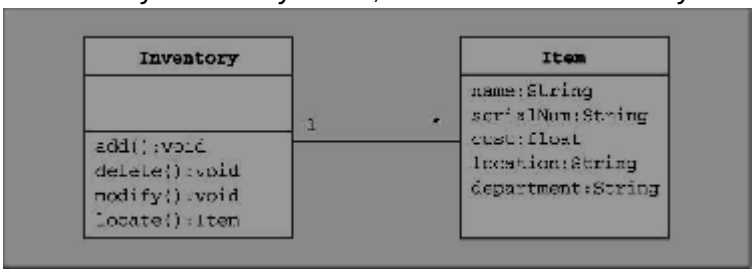
Figure 3.10 Association vs. link. A pair of objects can be instantiated at most once per association (except for bags and sequences).

Figure 3.11 Association vs. link. You can use multiple associations to model multiple links between the same objects.

Multiplicity of Associations

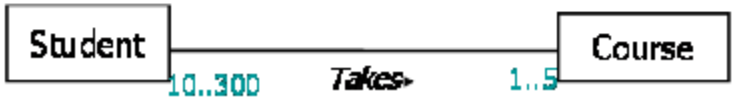
- Many-to-one
  - Bank has many ATMs, ATM knows only 1 bank

- One-to-many
  - Inventory has many items, items know 1 inventory



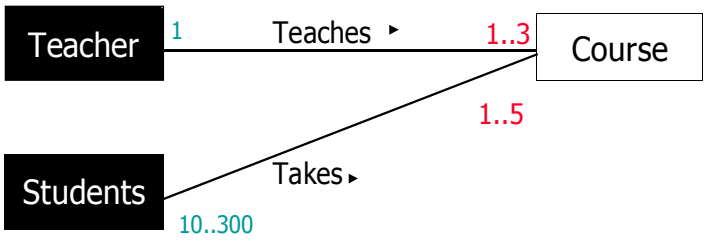
Association - Multiplicity

- A Student can take up to five Courses.
- Student has to be enrolled in at least one course.
- Up to 300 students can enroll in a course.
- A class should have at least 10 students.



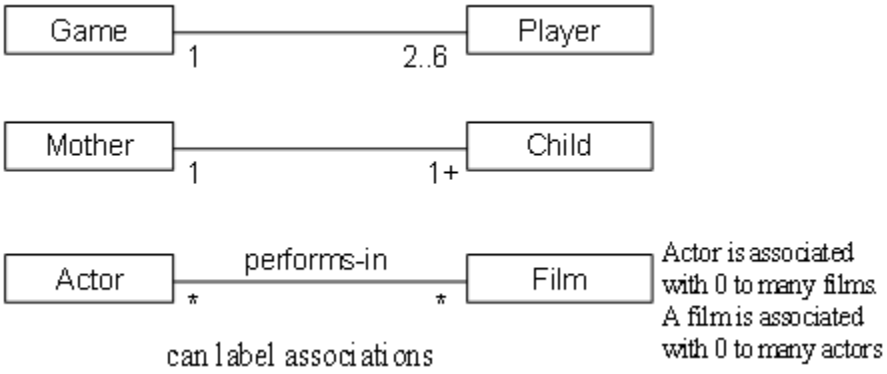
Association – Multiplicity

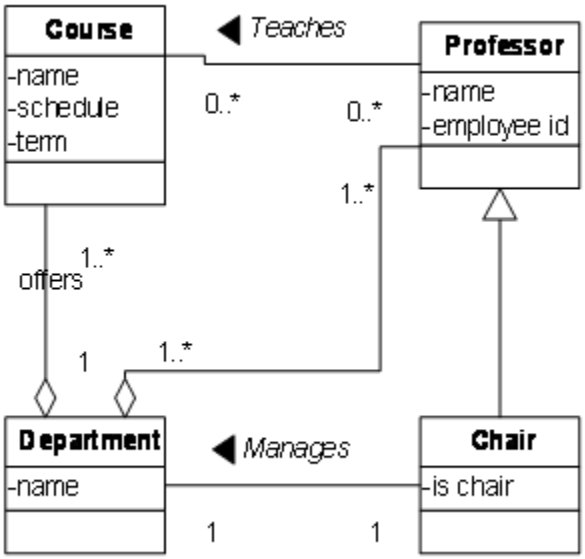
- A teacher teaches 1 to 3 courses (subjects)
- Each course is taught by only one teacher.
- A student can take between 1 to 5 courses.
- A course can have 10 to 300 students.



Multiplicity

- Multiplicity defines how many instances of type A can be associated with one instance of type B at some point





MULTIPLICITIES IN ASSOCIATIONS

min..max notation (related to at least min objects and at most max objects)	0..*	related to zero or more objects
	0..1	related to no object or at most one object
	1..*	related to at least one object
	1..1	related to exactly one object.
	3..5	related to at least three objects and at most five objects
short hand notation	1	same as 1..1
	*	same as 0..*

_**Note 2:** Multiplicity vs Cardinality.

- Multiplicity is a constraint on the size of a collection.
- Cardinality is a count of elements that are actually in a collection.

Therefore, multiplicity is a constraint on cardinality.

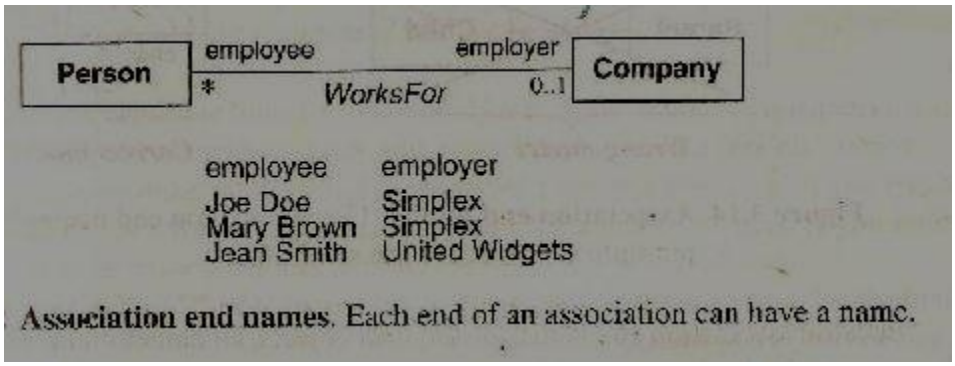
_**Note 3:** The literature often describes multiplicity as being "one" or "many", but more generally it is a subset of the non negative numbers.

Association end names

_Multiplicity implicitly refers to the ends of associations. For E.g. A one-to-many association has two ends –

- an end with a multiplicity of "one"
- an end with a multiplicity of "many"

You can not only assign a multiplicity to an association end, but you can give it a name as well.



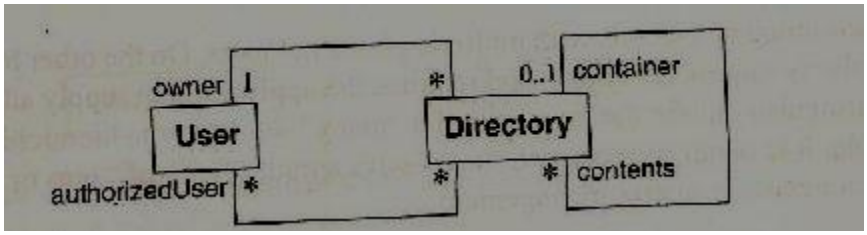
A person is an employee with respect to company.

A company is an employer with respect to a person.

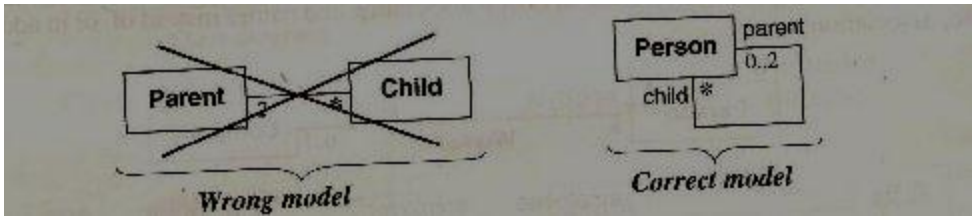
_Note 1: Association end names are optional.

_Note 2: Association end names are necessary for associations between two objects of the same class. They can also distinguish multiple associations between a pair of classes.

E.g. each directory has exactly one user who is an owner and many users who are authorized to use the directory. When there is only a single association between a pair of distinct classes, the names of the classes often suffice, and you may omit association end names.



_Note 3: Association end names let you unify multiple references to the same class. When constructing class diagrams you should properly use association end names and not introduce a separate class for each reference as below fig shows.



Sometimes, the objects on a "many" association end have an explicit order.

E.g. Workstation screen containing a number of overlapping windows. Each window on a screen occurs at most once. The windows have explicit order so only the top most windows are visible at any point on the screen.

_Ordering is an inherent part of association. You can indicate an ordered set of objects by writing "{ordered}" next to the appropriate association end.

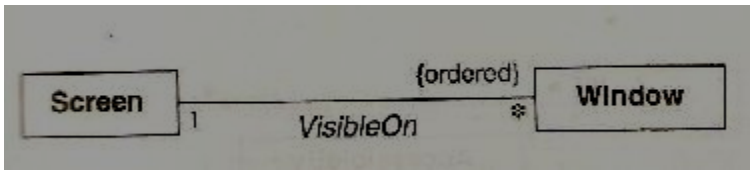


Fig: ordering sometimes occurs for "many" multiplicity

Bags and Sequences

_ Normally, a binary association has **at most one link** for a pair of objects.

_ However, you can permit **multiple links** for a pair of objects by annotating an 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.

Example:

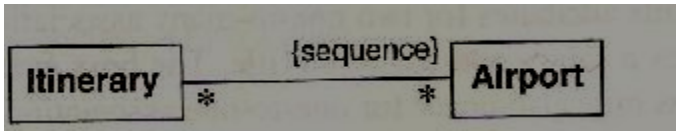


fig: an itinerary may visit multiple airports, so you should use {sequence} and not {ordered}

_ **Note:** {ordered} and {sequence} annotations are same, except that the first disallows duplicates and the other allows them.

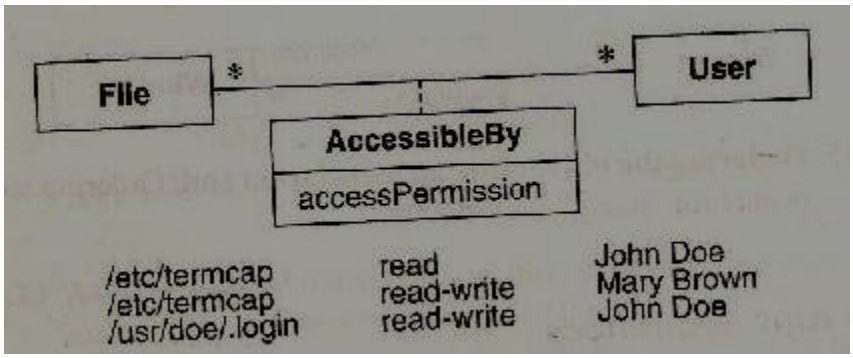
Association classes

_ An **association class** is an association that is also a class.

Like the links of an association, the instances of an association class derive identity from instances of the constituent classes.

Like a class, an association class can have attributes and operations and participate in associations.

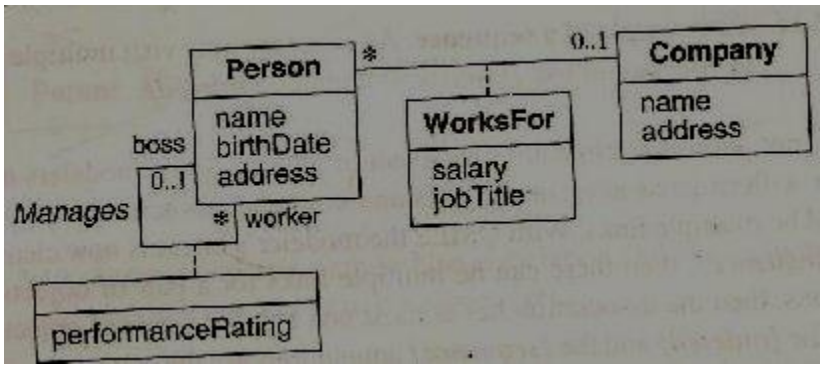
_ Ex:



_ **UML notation** for association class is a box attached to the association by a dashed line.

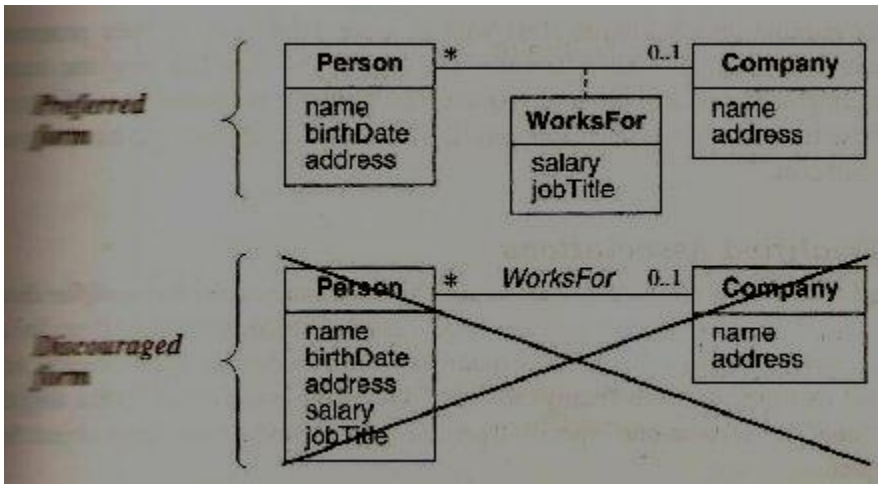
_ **Note:** Attributes for association class unmistakably belong to the link and cannot be ascribed to either object. In the above figure, accessPermission is a joint property of File and user cannot be attached to either file or user alone without losing information.

_Below figure presents attributes for two one-to-many relationships. Each person working for a company receives a salary and has job title. The boss evaluates the performance of each worker. Attributes may also occur for one-to-one associations.

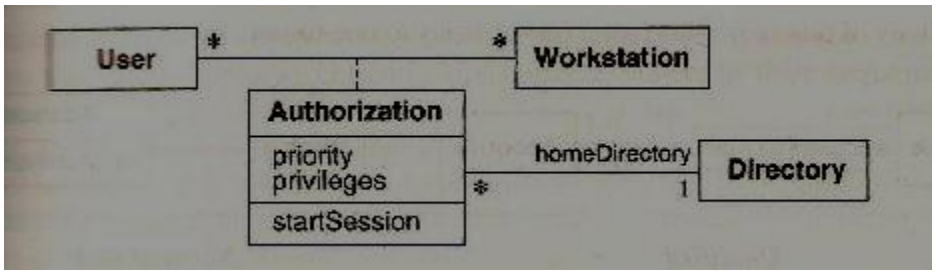


_**Note 1:** Figure shows how it's possible to fold attributes for one-to-one and one-to-many associations into the class opposite a "one" end. This is not possible for many-to-many associations.

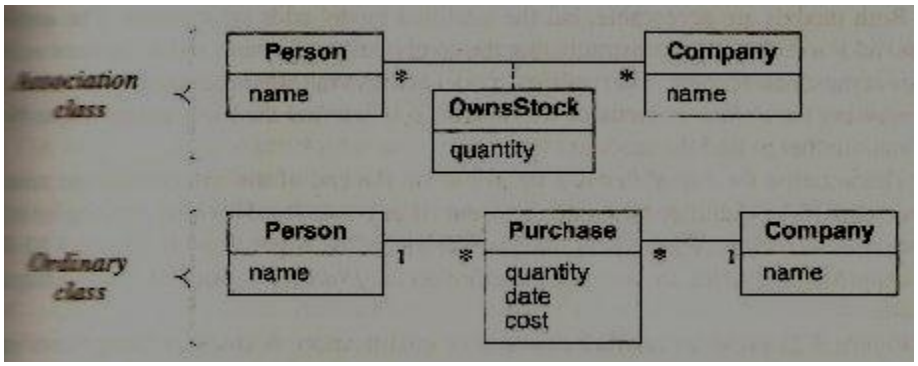
As a rule, you should not fold such attributes into a class because the multiplicity of the association may change.



_**Note 2:** An association class participating in an association.



_**Note 3:** Association class vs ordinary class.



eg:

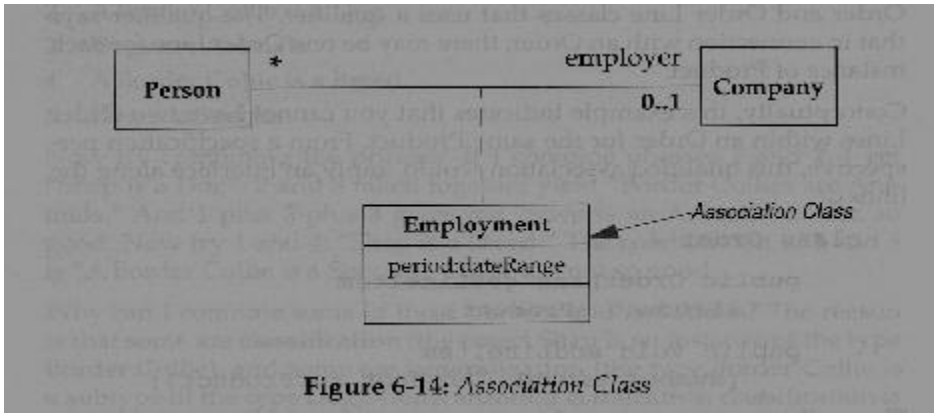
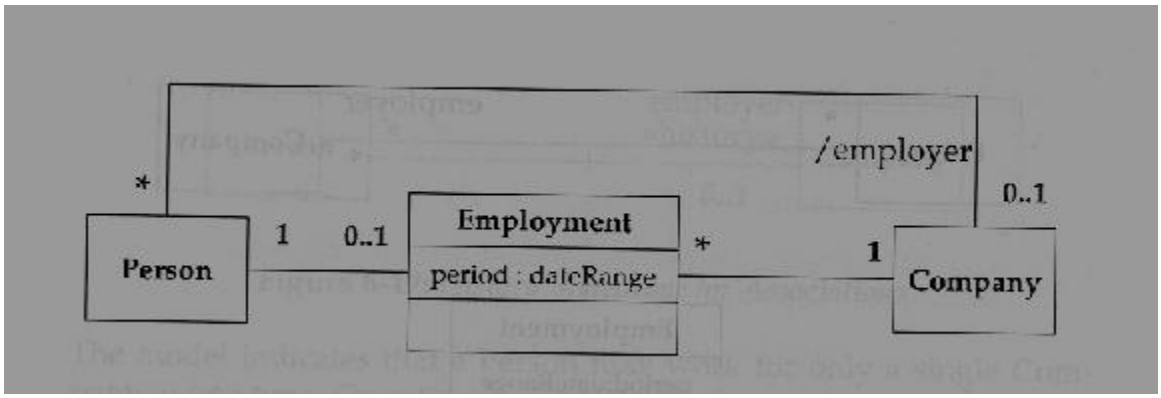


Figure 6-14: *Association Class*



Qualified associations

_ A **Qualified Association** is an association in which an attribute called the **qualifier** disambiguates the objects for a "many" association ends. 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".

_ **Ex 1:** qualifier for associations with one to many multiplicity. A bank services multiple accounts. An account belongs to single bank. Within the context of a bank, the Account Number specifies a unique account. Bank and account are classes, and Account Number is a qualifier. Qualification reduces effective multiplicity of this association from one-to-many to one-to-one.

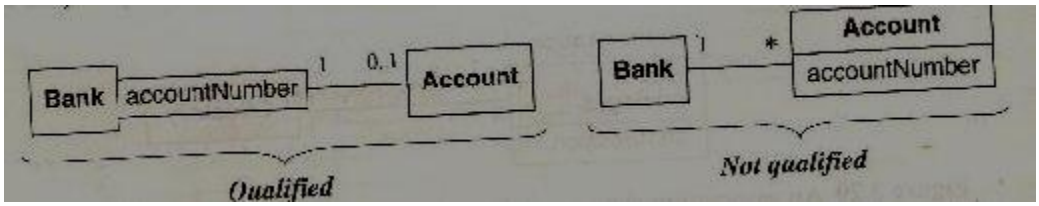
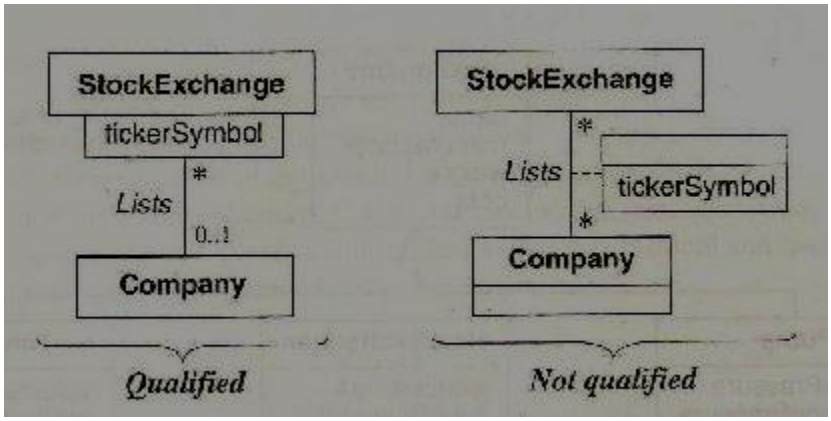
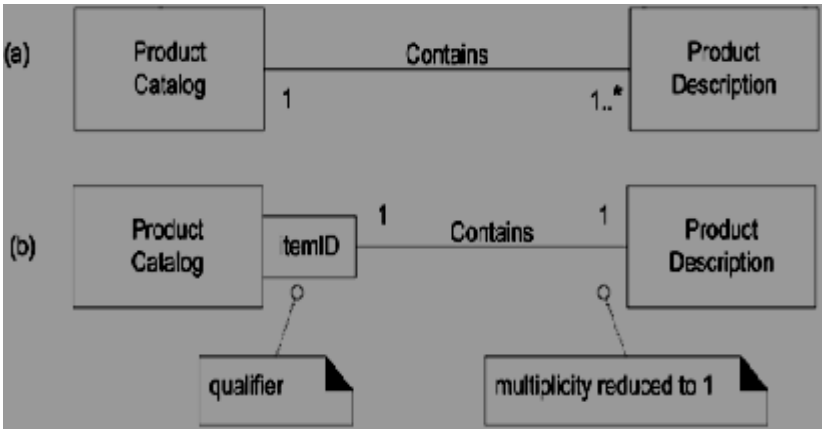


Fig: qualification increases the precision of a model. (note: however, both are acceptable)

_Ex 2: a stock exchange lists many companies. However, it lists only one company with a given ticker symbol. A company maybe listed on many stock exchanges, possibly under different symbols.



Eg 3: Qualified Association



GENERALIZATION AND INHERITANCE

_**Generalization** is the relationship between a class (the superclass) and one or more variations of the class (the subclasses). Generalization organizes classes by their similarities and differences, structuring the description of 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.

_There can be **multiple levels** of generalization.

_Fig(a) and Fig(b) (given in the following page) shows examples of generalization.

_Fig(a) – Example of generalization for equipment.

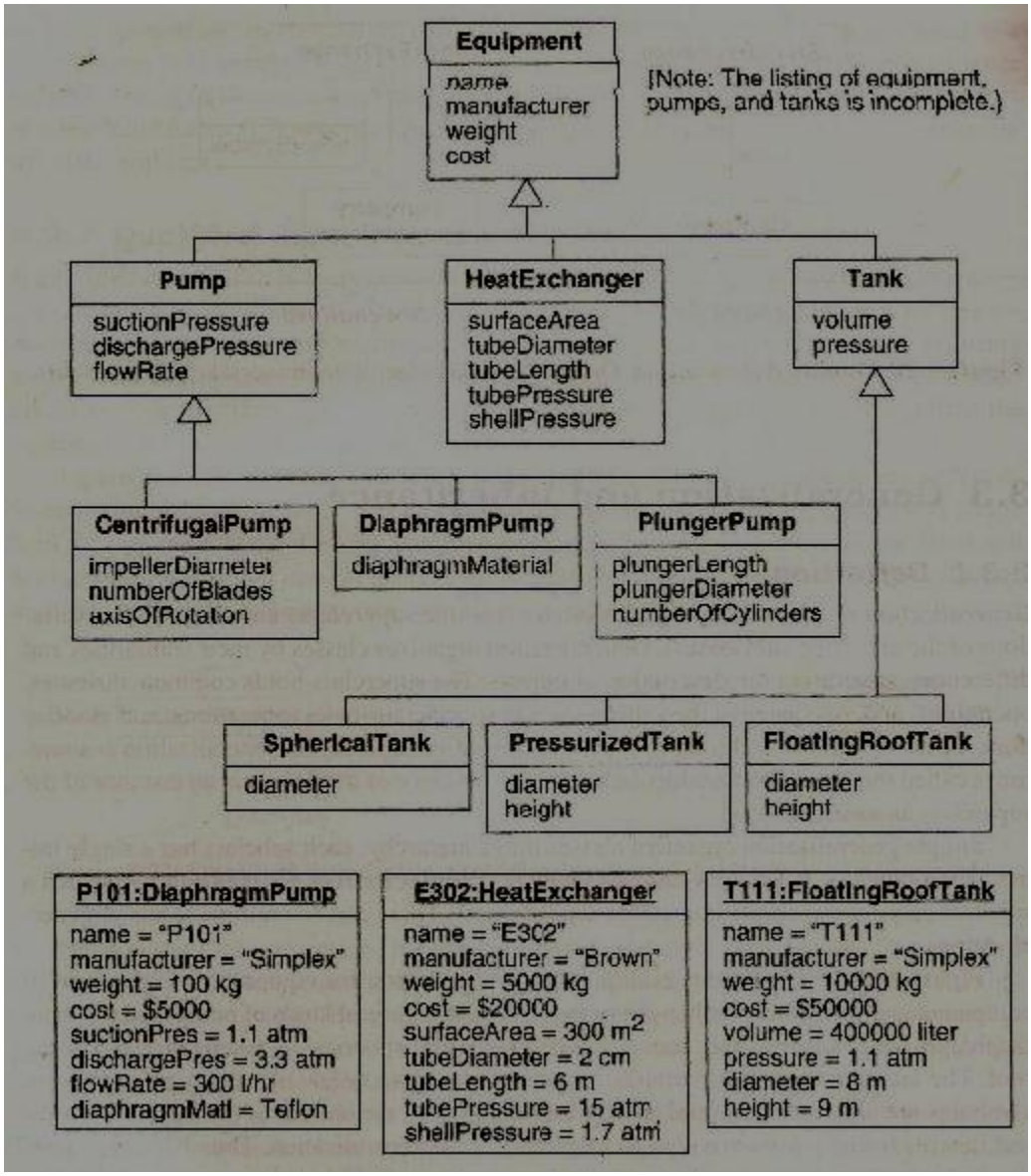
Each object inherits features from one class at each level of generalization.

_UML convention used:

Use large hollow arrowhead to denote generalization. The arrowhead points to superclass.

_Fig(b) – inheritance for graphic figures.

The word written next to the generalization line in the diagram (i.e dimensionality) is a generalization set name. A generalization set name is an enumerated attribute that indicates which aspect of an object is being abstracted by a particular generalization. It is optional.


Equipment
name
manufacturer
weight
cost
[Note: The listing of equipment, pumps, and tanks is incomplete.)
Pump
suctionPressure
dischargePressure
flowRate
HeatExchanger
surfaceArea
tubeDiameter
tubeLength
tubePressure
shellPressure
Tank
volume
pressure
CentrifugalPump
impellerDiameter
numberOfBlades
axisOfRotation
DiaphragmPump
diaphragmMaterial
PlungerPump
plungerLength
plungerDiameter
numberOfCylinders
SphericalTank
diameter
PressurizedTank
diameter
height
FloatingRoofTank
diameter
height
P101:DiaphragmPump
name = "P101"
manufacturer = "Simplex"
weight = 100 kg
cost = $5000
suctionPres = 1.1 atm
dischargePres = 3.3 atm
flowRate = 300 l/hr
diaphragmMatl = Teflon
E302:HeatExchanger
name = "E302"
manufacturer = "Brown"
weight = 5000 kg
cost = $20000
surfaceArea = 300 m2
tubeDiameter = 2 cm
tubeLength = 6 m
tubePressure = 15 atm
shellPressure = 1.7 atm
T111:FloatingRoofTank
name = "T111"
manufacturer = "Simplex"
weight = 10000 kg
cost = $50000
volume = 400000 liter
pressure = 1.1 atm
diameter = 8 m
height = 9 m

Fig(a)

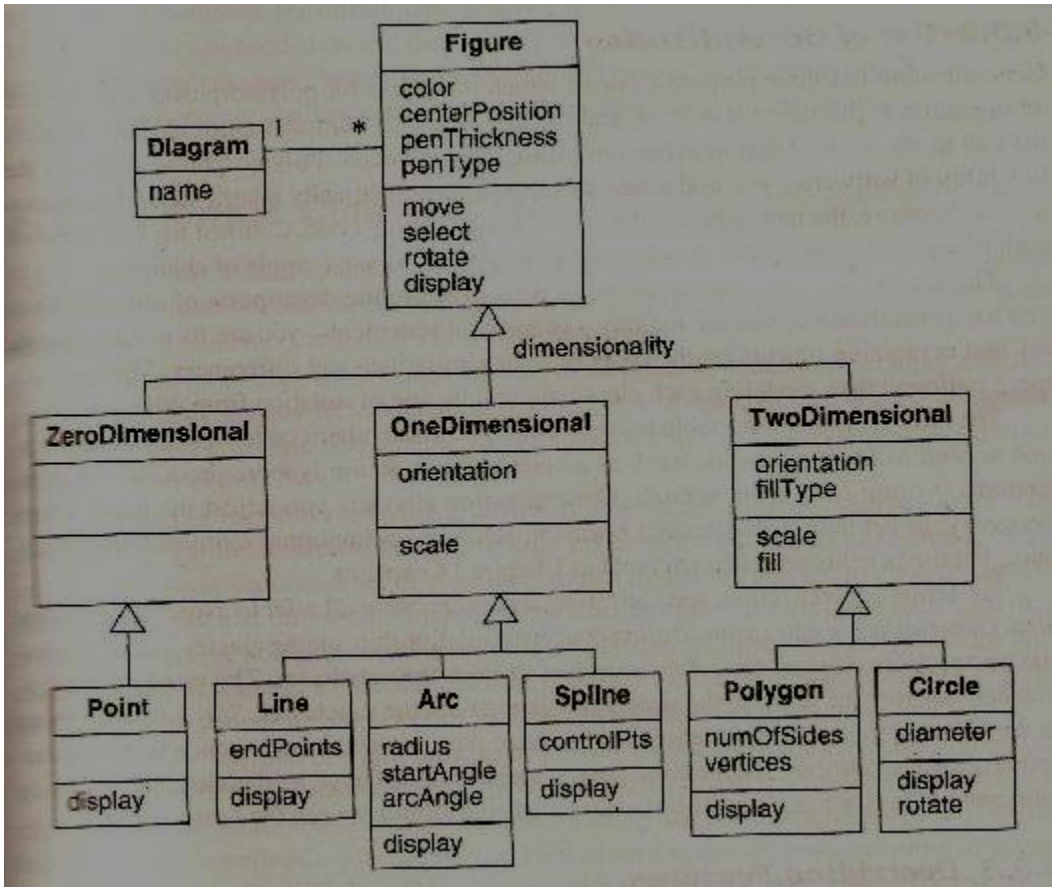


Fig (b)

'move**',** **'**select**',** 'rotate', and 'display' are operations that all subclasses inherit.

'scale**'** applies to one-dimensional and two-dimensional figures.

'fill' applies only to two-dimensional figures.

_**Use of generalization:** Generalization has three purposes –

1. **To support polymorphism:** You can call an operation at the superclass level, and the OO language complier automatically resolves the call to the method that matches the calling object's class.
2. **To structure the description of objects:** i.e to frame a taxonomy and organizing objects on the basis of their similarities and differences.
3. **To enable reuse of code:** Reuse is more productive than repeatedly writing code from scratch.

_**Note:** The terms generalization, specialization and inheritance all refer to aspects of the same idea.

Overriding features

_A subclass may override a superclass feature by defining a feature with the same name. The overriding feature (subclass feature) refines and replaces the overridden feature (superclass feature) .

_Why override feature?

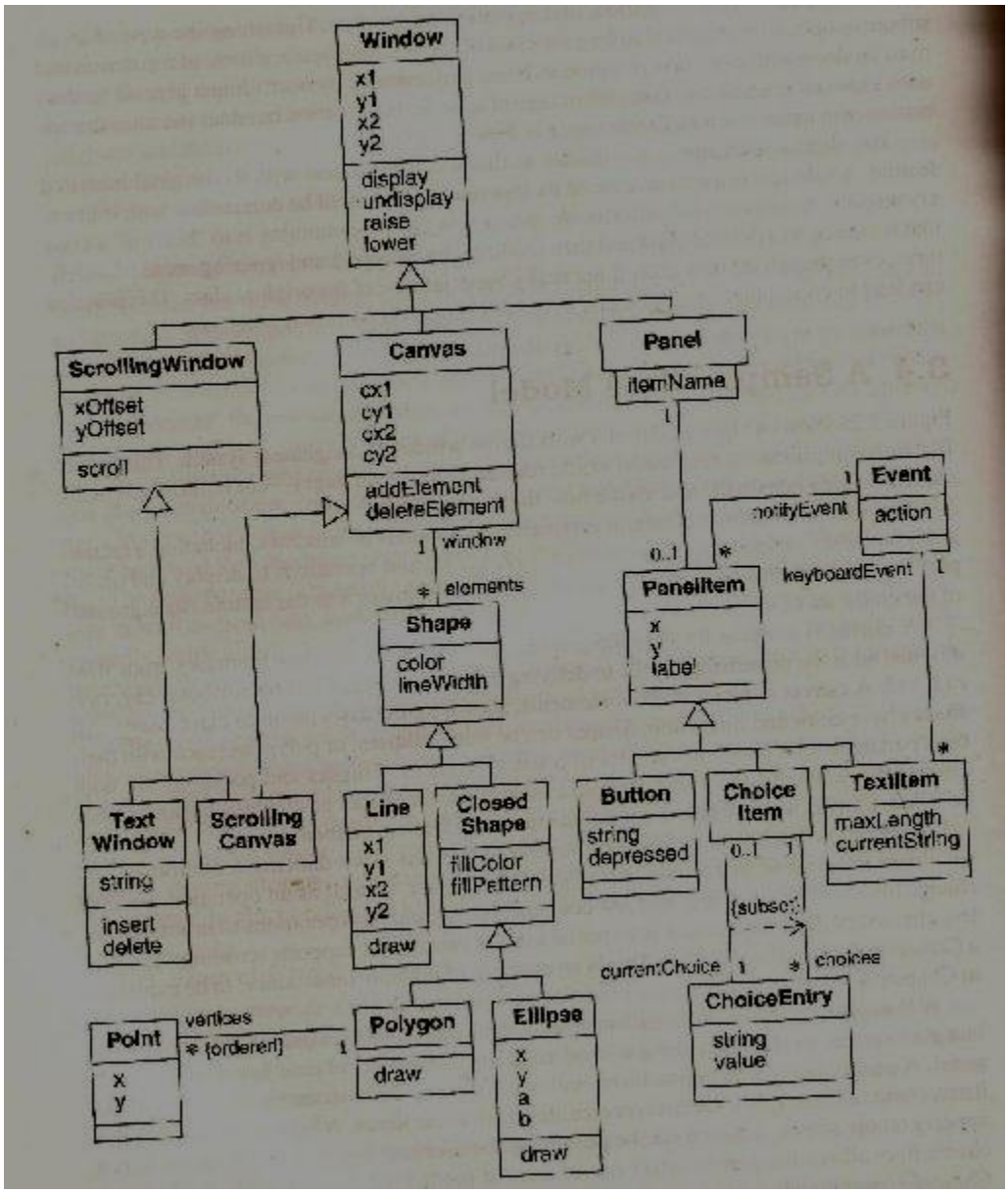
- To specify behavior that depends on subclass.
- To tighten the specification of a feature.

- To improve performance.

_ In fig(b) (previous page) each leaf subclasses had overridden 'display' feature.

_ **Note:** You may override methods and default values of attributes. You should never override the signature, or form of a feature.

A SAMPLE CLASS MODEL



NAVIGATION OF CLASS MODELS

_ Class models are useful for more than just data structure. In particular, **navigation of class model** lets you express certain behavior. Furthermore, navigation exercises a class model and uncovers hidden flaws and omission, which you can then repair.

_ UML incorporates a language that can be used for navigation, the **object constraint language(OCL).**

–

OCL constructs for traversing class models

OCL can traverse the constructs in class models.

1. **Attributes:** You can traverse from an object to an attribute value.

   Syntax: source object followed by dot and then attribute name.

   Ex: aCreditCardAccount.maximumcredit

2. **Operations: Y**ou can also invoke an operation for an object or collection of objects. Syntax: source object or object collection, followed by dot and then the operation followed by parenthesis even if it has no arguments. OCL has special operations that operate on entire collections (as opposed to operating on each object in a collection). Syntax for collection operation is: source object collection followed by "->", followed by the operation.

3. **Simple associations:** Dot notation is also used to traverse an association to a target end. Target end maybe indicated by an association end name, or class name (if there is no ambiguity).

   Ex: refer fig in next page.

   - aCustomer.MailingAddress yields a set of addresses for a customer (the target end has "many" multiplicity).
   - aCreditCardAccount.MailingAddress yields a single address(the target end has multiplicity of "one").

4. **Qualified associations:** The expression aCreditCardAccount.Statement [30 November 1999] finds the statement for a credit card account with the statement date of November 1999. The syntax is to enclose the qualifier value in brackets.

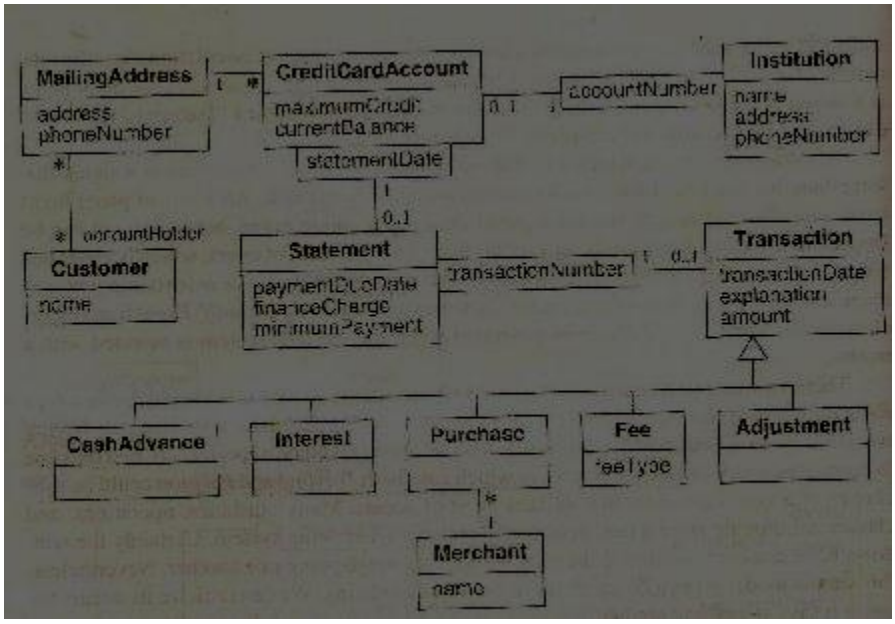
5. **Associations classes:** Given a link of an association class, you can find the constituent objects and vice versa.

6. **Generalization:** Traversal of a generalization hierarchy is implicit for the OCL notation.

7. **Filters:** Most common filter is 'select' operation.

   Ex: aStatement.Transaction->select(amount>$100).

Examples of OCL expressions



_Write an OCL expression for–

1. What transactions occurred for a credit card account within a time interval?

Soln: aCreditCardAccount.Statement.Transaction ->
select(aStartDate<=TransactionDate and
TransactionDate<=anEndDate)

2. What volumes of transactions were handled by an institution in the last year?

Soln: anInstitution.CreditCardAccount.Statement.Transaction ->
select(aStartDate<=TransactionDate and TransactionDate<=anEndDate).amount->sum( )

3. What customers patronized a merchant in the last year by any kind of credit card?

Soln: aMerchant.Purchase -> select(aStartDate<=TransactionDate andtransactionDate<=anEndDate).Statement.CreditCardAccount.MailingAddress.Customer ->asset( )

4. How many credit card accounts does a customer currently have?

Soln: aCustomer.MailingAddress.CreditCardAccount -> size( )

5. What is the total maximum credit for a customer for all accounts?

Soln: acustomer.MailingAddress.CreditCardAccount.Maximumcredit -> sum( )

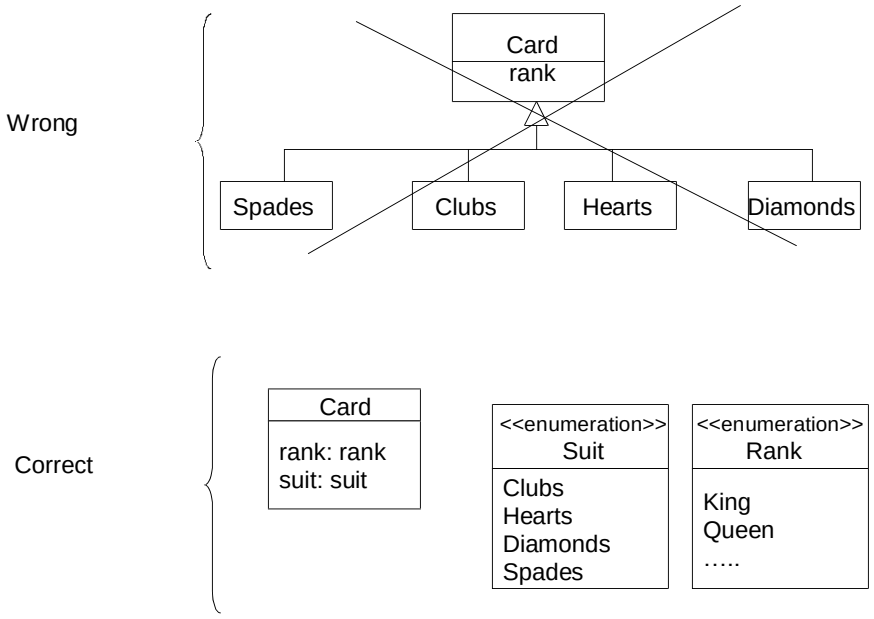
Advanced Class Modeling

2.1 Advanced object and class concepts

2.1.1 Enumerations

A data type is a description of values, includes numbers, strings, enumerations
Enumerations: A Data type that has a finite set of values.

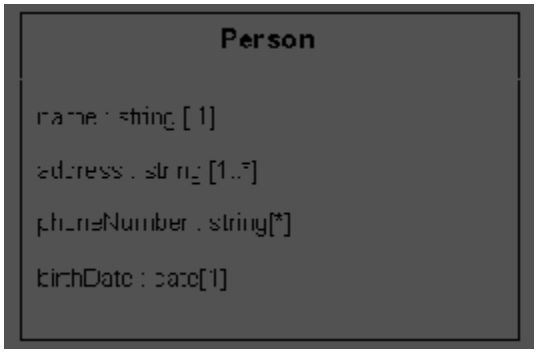
- When constructing a model, we should carefully note enumerations, because they often occur and are important to users.
- Enumerations are also significant for an implantation; we may display the possible values with a pick list and you must restrict data to the legitimate values.
- Do not use a generalization to capture the values of an Enumerated attribute.
- An Enumeration is merely a list of values; generalization is a means for structuring the description of objects.
- Introduce generalization only when at least one subclass has significant attributes, operations, or associations that do not apply to the superclass.
- In the UML an enumeration is a data type.
- We can declare an enumeration by listing the keyword *enumeration* in guillemets (<< >>) above the enumeration name in the top section of a box. The second section lists the enumeration values. (Diagram page number 60 fig 4.1)
- Eg: Boolean type= { TRUE, FALSE}
- Eg: figure.pentype _____ - - - - - --------
- Two diml.filltype 



Modeling enumerations. Do not use a generalization to capture the values of an enumerated attribute

2.1.2 Multiplicity

- Multiplicity is a collection on the cardinality of a set, also applied to attributes (database application).
- Multiplicity of an attribute specifies the number of possible values for each instantiation of an attribute. i.e., whether an attribute is mandatory ([1]) or an optional value ([0..1] or * i.e., null value for database attributes) .
- Multiplicity also indicates whether an attribute is single valued or can be a collection.



2.1.3 Scope

- Scope indicates if a feature applies to an object or a class.
- An underline distinguishes feature with class scope (static) from those with object scope.
- Our convention is to list attributes and operations with class scope at the top of the attribute and operation boxes, respectively.

- It is acceptable to use an attribute with class scope to hole the **extent** of a class (the set of objects for a class) - this is common with OO databases. Otherwise, you should avoid attributes with class scope because they can lead to an inferior model.
- It is better to model groups explicitly and assigns attributes to them.
- In contrast to attributes, it is acceptable to define operations of class scope. The most common use of class-scoped operations is to create new instances of a class, sometimes for summary data as well.

2.1.4 Visibility

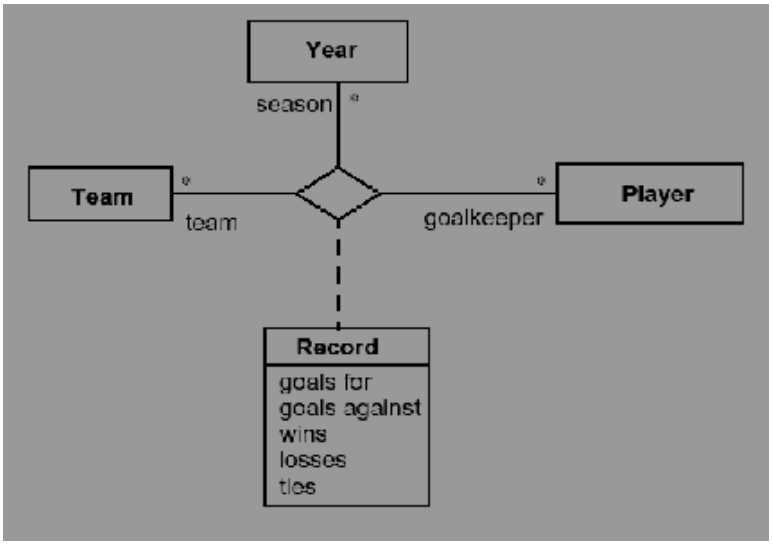
- Visibility refers to the ability of a method to reference a feature from another class and has the possible values of *public, protected, private,* and *package.*
- Any method can access **public** features.
- Only methods of the containing class and its descendants via inheritance can access **protected** features.
- Only methods of the containing class can access **private** features.
- Methods of classes defined in the same package as the target class can access **package** features
- The UML denotes visibility with a prefix. **"+"→ public, "-"→ private, "#"→protected, "~"→ package.** Lack of a prefix reveals no information about visibility.
- Several issues to consider when choosing visibility are
  - **Comprehension**: understand all public features to understand the capabilities of a class. In contrast we can ignore private, protected, package features – they are merely an implementation convince.
  - **Extensibility**: many classes can depend on public methods, so it can be highly disruptive to change their signature. Since fewer classes depend on private, protected, and package methods, there is more latitude to change them.
  - **Context**: private, protected, and package methods may rely on preconditions or state information created by other methods in the class. Applied out of context, a private method may calculate incorrect results or cause the object to fail.

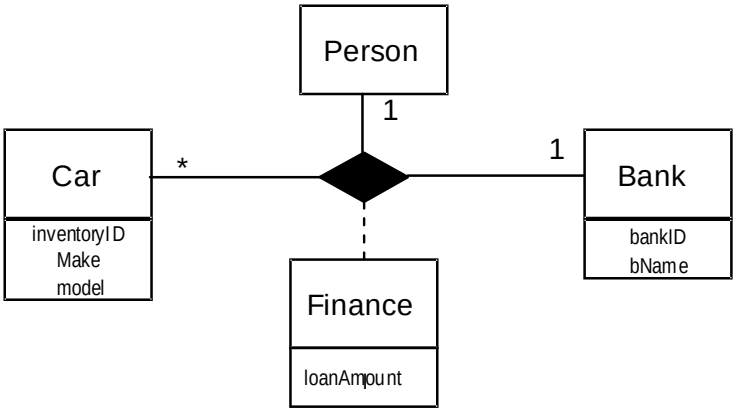
2.2 Associations ends

- Association End is an end of association.
- A binary association has 2 ends; a ternary association has 3 ends.

2.3 N-ary Association

- We may occasionally encounter n-ary associations (association among 3 or more classes). But we should try to avoid n-ary associations- most of them can be decomposed into binary associations, with possible qualifiers and attributes.



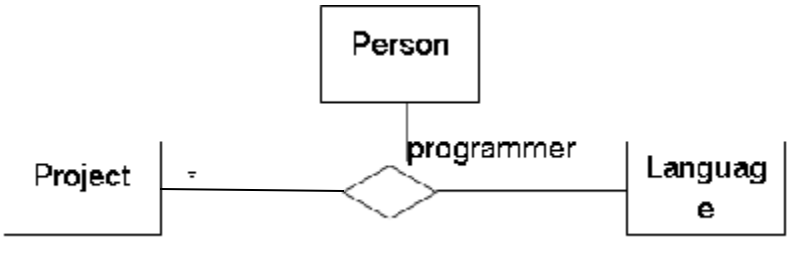


-
- The UML symbol for n-ary associations is a diamond with lines connecting to related classes. If the association has a name, it is written in italics next to the diamond.
- The OCL does not define notation for traversing n-ary associations.

- A typical programming language cannot express n-ary associations. So, promote n-ary associations to classes. Be aware that you change the meaning of a model, when you promote n-ary associations to classes.
- An n-ary association enforces that there is at most one link for each combination.

Eg:

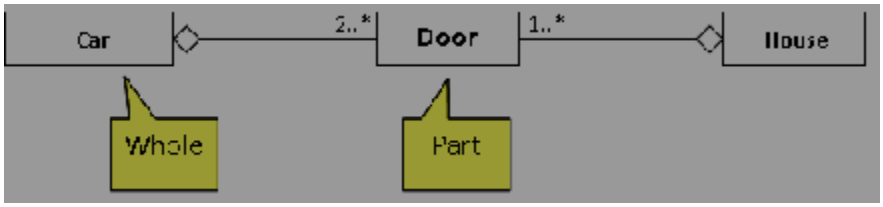
Class diagram



Instance diagram *see prescribed text book page no. 65 and fing no. 4.6*

2.4 Aggregation

- Aggregation is a strong form of association in which an aggregate object is made of constituent parts.
- Constituents are the parts of 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.
- 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 (if A is part of B and B is part of C, then A is part of C) and antisymmetric (if A is part of B then B is not part of A)



2.4.1 Aggregation versus Association

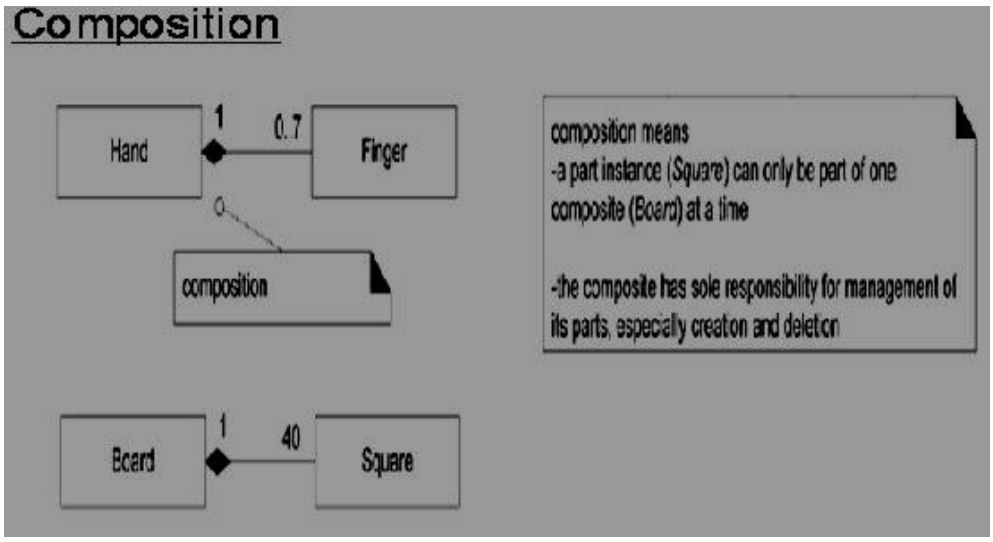
- Aggregation is a special form of association, not an independent concept.
- Aggregation adds semantic connotations.
- If two objects are tightly bound by a part-whole relationship, it is an aggregation. If the two objects are usually considered as independent, even though they may often be linked, it is an association.
- Aggregation is drawn like association, except a small (hollow) diamond indicates the assembly end.

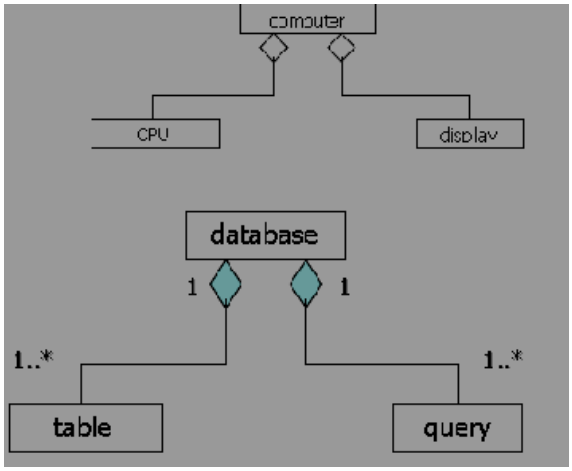
- The decision to use aggregation is a matter of judgment and can be arbitrary.

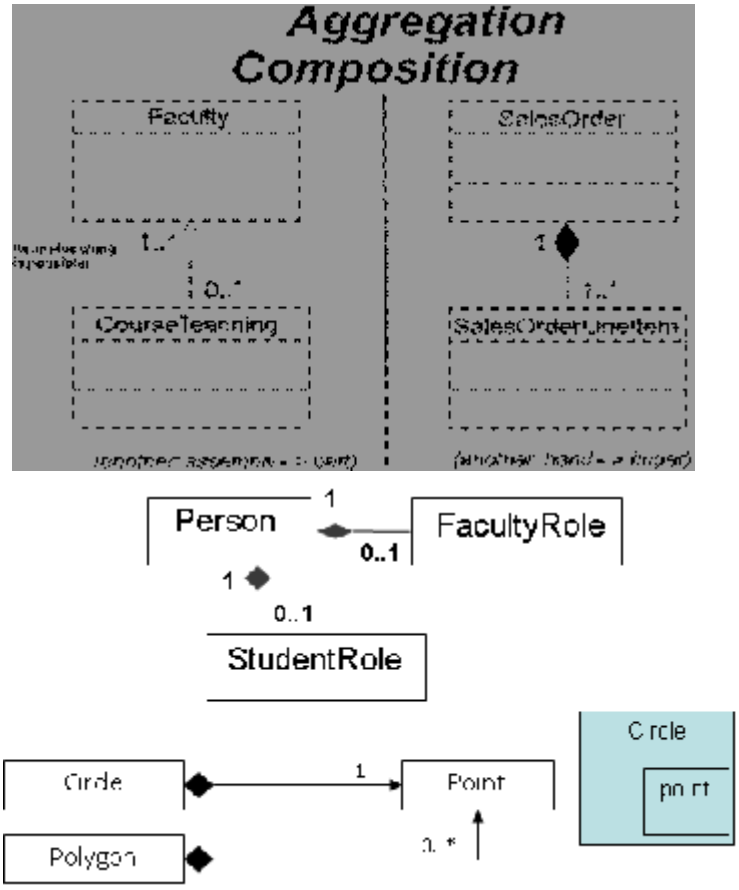
2.4.2 Aggregation versus Composition

- The UML has 2 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 constitute part can belong to at most one assembly.
- Once a constitute 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.
- Notation for composition is a small solid diamond next to the assembly class.

Eg: see text book examples also







2.4.3 Propagation of Operations

- Propagation (triggering) is the automatic application of an operation to a network of objects when the operation is applied to some starting object.
- For example, moving an aggregate moves its parts; the move operation propagates to the parts.
- Provides concise and powerful way of specifying a continuum behavior.
- Propagation is possible for other operations including save/restore, destroy, print, lock, display.
- Notation (not an UML notation): a small arrow indicating the direction and operation name next to the affected association.

  Eg: see page no: 68 fig: 4.11

2.5 Abstract Classes

- Abstract class is a class that has no direct instances but whose descendant classes have direct instances.
- A concert class is a class that is insatiable; that is, it can have direct instances.
- A concrete class may have abstract class.
- Only concrete classes may be leaf classes in an inheritance tree.

  Eg: see text book page no: 69, 70 fig: 4.12, 4.13,4.14

- In UML notation an abstract class name is listed in an italic (or place the keyword {abstract} below or after the name).
- We can use abstract classes to define the methods that can be inherited by subclasses.
- Alternatively, an abstract class can define the signature for an operation with out supplying a corresponding method. We call this an abstract operation.
- Abstract operation defines the signature of an operation for which each concrete subclass must provid4 its own implementation.
- A concrete class may not contain abstract operations, because objects of the concrete class would have undefined operations.

2.6 Multiple Inheritance

- Multiple inheritance permits a class to have more than one superclass and to inherit features from all parents.
- We can mix information from 2 or more sources.
- This is a more complicated from of generalization than single inheritance, which restricts the class hierarchy to a tree.
- The advantage of multiple inheritance is greater power in specifying classes and an increased opportunity for reuse.
- The disadvantage is a loss of conceptual and implementation simplicity.
- The term multiple inheritance is used somewhat imprecisely to mean either the conceptual relationship between classes or the language mechanism that implements that relationship.

2.6.1 Kinds of Multiple Inheritance

- The most common form of multiple inheritance is from sets of disjoint classes. Each subclass inherits from one class in each set.
- The appropriate combinations depend on the needs of an application.
- Each generalization should cover a single aspect.
- We should use multiple generalizations if a class can be refined on several distinct and independent aspects.
- A subclass inherits a feature from the same ancestor class found along more than one path only once; it is the same feature.
- Conflicts among parallel definitions create ambiguities that implementations must resolve. In practice, avoid such conflicts in models or explicitly resolve them, even if a particular language provides a priority rule for resolving conflicts.
- The UML uses a constraint to indicate an overlapping generalization set; the notation is a dotted line cutting across the affected generalization with keywords in braces.
  Eg: see text book page no: 71,72 fig: 4.15,4.16

2.6.2 Multiple Classification

- An instance of a class is inherently an instance of all ancestors of the class.
- For example, an instructor could be both faculty and student. But what about a Harvard Professor taking classes at MIT? There is no class to describe the

combination. This is an example of multiple classification, in which one instance happens to participate in two overlapping classes.

Eg: see text book page no: 73 fig: 4.17

2.6.3 Workarounds

- Dealing with lack of multiple inheritance is really an implementation issue, but early restructuring of a model is often the easiest way to work around its absence.
- Here we list 2 approaches for restructuring techniques (it uses delegation)
- Delegation is an implementation mechanism by which an object forwards an operation to another object for execution.

1. **Delegation using composition of parts:** Here we can recast a superclass with multiple independent generalization as a composition in which each constituent part replaces a generalization. This is similar to multiple classification. This approach replaces a single object having a unique ID by a group of related objects that compose an extended object. Inheritance of operations across the composition is not automatic. The composite must catch operations and delegate them to the appropriate part.

   In this approach, we need not create the various combinations as explicit classes. All combinations of subclasses from the different generalization are possible.
2. **Inherit the most important class and delegate the rest:**
   Fig 4.19 preserves identity and inheritance across the most important generalization. We degrade the remaining generalization to composition and delegate their operations as in previous alternative.
3. **Nested generalization:** this approach multiplies out all possible combinations. This preserves inheritance but duplicates declarations and code and violets the spirit of OO programming.
4. **Superclasses of equal importance:** if a subclass has several superclasses, all of equal importance, it may be best to use delegation and preserve symmetry in the model.
5. **Dominant superclass:** if one superclass clearly dominates and the others are less important, preserve inheritance through this path.
6. **Few subclasses:** if the number of combinations is small, consider nested generalization. If the number of combinations is large, avoid it.
7. **Sequencing generalization sets:** if we use generalization, factor on the most important criterion first, the next most important second, and so forth.
8. **Large quantities of code:** try to avoid nested generalization if we must duplicate large quantities of code.
9. **Identity:** consider the importance of maintaining strict identity. Only nested generalization preserves this.

2.7 Metadata

- Metadata is data that describes other data. For example, a class definition is a metadata.
- Models are inherently metadata, since they describe the things being modeled (rather than being the things).
- Many real-world applications have metadata, such as parts catalogs, blueprints, and dictionaries. Computer-languages implementations also use metadata heavily.
- We can also consider classes as objects, but classes are meta-objects and not real-world objects. Class descriptor object have features, and they in turn have their own classes, which are called ***metaclasses.***

  Eg: see text book page no: 75 fig: 4.21

2.8 Reification

- Reification is the promotion of something that is not an object into an object.
- Reification is a helpful technique for Meta applications because it lets you shift the level of abstraction.
- On occasion it is useful to promote attributes, methods, constraints, and control information into objects so you can describe and manipulate them as data.
- As an example of reification, consider a database manager. A developer could write code for each application so that it can read and write from files. Instead, for many applications, it is better idea to reify the notion of data services and use a database manager. A database manager has abstract functionality that provides a general-purpose solution to accessing data reliably and quickly for multiple users.

  Eg: see text book page no: 75 fig: 4.22

2.9 Constraints

- Constraint is a condition involving model elements, such as objects, classes, attributes, links, associations, and generalization sets.
- A Constraint restricts the values that elements can assume by using OCL.

2.9.1 Constraints on objects

Eg: see text book page no: 77 fig: 4.23

2.9.2 Constraints on generalization sets

- Class models capture many Constraints through their very structure. For example, the semantics of generalization imply certain structural constraints.
- With single inheritance the subclasses are mutually exclusive. Furthermore, each instance of an abstract superclass corresponds to exactly one subclass instance. Each instance of a concrete superclass corresponds to at most one subclass instance.
- The UML defines the following keyword s for generalization.
  - **Disjoint:** The subclasses are mutually exclusive. Each object belongs to exactly one of the subclasses.
  - **Overlapping:** The subclasses can share some objects. An object may belong to more than one subclass.

- **Complete:** The generalization lists all the possible subclasses.
- **Incomplete:** The generalization may be missing some subclasses.

2.9.3 Constraints on Links

- Multiplicity is a constraint on the cardinality of a set. Multiplicity for an association restricts the number of objects related to a given object.
- Multiplicity for an attribute specifies the number of values that are possible for each instantiation of an attribute.
- Qualification also constraints an association. A qualifier attribute does not merely describe the links of an association but is also significant in resolving the "many" objects at an association end.
- An association class implies a constraint. An association class is a class in every right; for example, it can have attribute and operations, participate in associations, and participate in generalization. But an association class has a constraint that an ordinary class does not; it derives identity from instances of the related classes.
- An ordinary association presumes no particular order on the object of a "many" end. The constraint {ordered} indicates that the elements of a "many" association end have an explicit order that must be preserved.

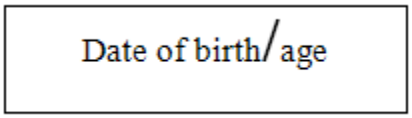
  Eg: see text book page no: 78 fig: 4.24

2.9.4 Use of constraints

- It is good to express constraints in a declarative manner. Declaration lets you express a constraint's intent, without supposing an implementation.
- Typically, we need to convert constraints to procedural form before we can implement them in a programming language, but this conversion is usually straightforward.
- A "good" class model captures many constraints through its structure. It often requires several iterations to get the structure of a model right from the prospective of constraints. Enforce only the important constraints.
- The UML has two alternative notations for constraints; either delimit a constraint with braces or place it in a "dog-earned" comment box. We can use dashed lines to connect constrained elements. A dashed arrow can connect a constrained element to the element on which it depends.

2.10. Derived Data

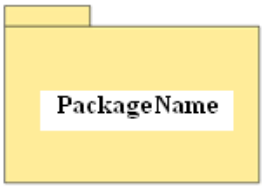
- A derived element is a function of one or more elements, which in turn may be derived. A derived element is redundant, because the other elements completely determine it. Ultimately, the derivation tree terminates with base elements. Classes, associations, and attributes may be derived. The notation for a derived element is a slash in front of the element name along with constraint that determines the derivation.



- A class model should generally distinguish independent base attributes from dependent derived attributes.
  Eg: see text book page no: 79 fig: 4.25

2.11 Packages

- A package is a group of elements (classes, association, generalization, and lesser packages) with a common theme.
- A package partitions a model, making it easier to understand and manage.
- A package partitions a model making it easier to understand and manage. Large applications my require several tiers of packages.
- Packages form a tree with increasing abstraction toward the root, which is the application, the top-level package.
- Notation for pakage is a box with a tab.



❖ Tips for devising packages

- Carefully delineate each packages's scope
- Define each class in a single package
- Make packages cohesive.