MODULE-2 **8 Hours**

INTRODUCTION, MODELING CONCEPTS, CLASS MODELING: 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. Class Modeling: Object and class concepts; Link and associations concepts; Generalization and inheritance; A sample class model; Navigation of class models;

Chapter – 1 INTRODUCTION, MODELING CONCEPTS, CLASS MODELING:

- What is object orientation?
- What is OO development?
- OO themes
- Evidence for usefulness of OO development
- OO modeling history
- Modeling
- Abstraction
- The tree models
- Objects and class concepts
- Link and association concepts
- Generalization and inheritance
- A sample class model
- Navigation of class models
- Practical tips

INTRODUCTION

Object oriented modeling and design is to learn how to apply object -oriented concepts to all the stages of the software development life cycle. Object oriented models are useful for understanding problems, communication with application experts, modeling enterprises, preparing documentation, and

designing programs and databases.

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 (Object Oriented) 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 approach

- 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 chessetc.

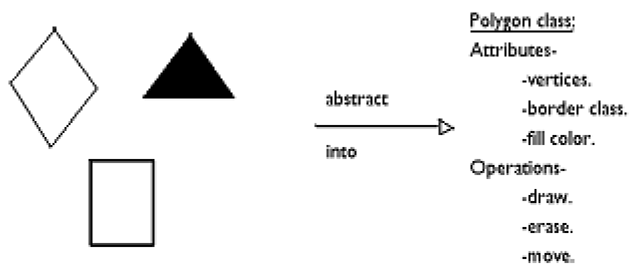
  Objects can be concrete (such as a file in a file system) or conceptual (such as scheduling policy in a multiprocessing OS). Each object has its own inherent identity. (i.e two objects are distinct even if all their attribute values (such as name and size areidentical).

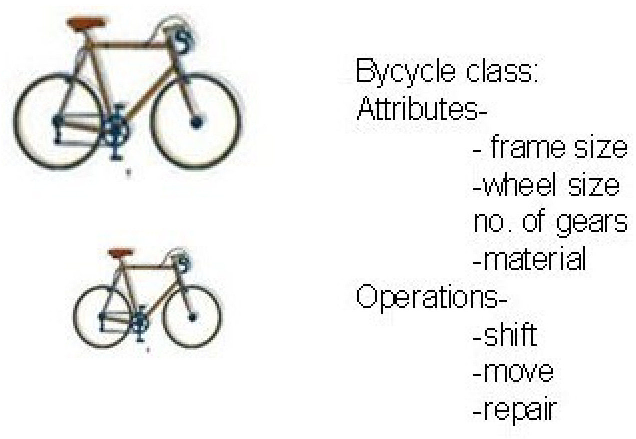
  In programming languages, an object is referenced by a unique handle by which it can be referenced (array index etc.,).

Classification:

- **Classification** means that objects with the same data structure (attributes) and behavior (operations) are grouped into aclass.
  - E.g. paragraph, monitor, chess piece.
  - Each object is said to be an instance of itsclass.
- Fig below shows objects and classes: Each class describes a possibly infinite set of individualobjects. An object contains an implicit reference to its own class; it "knows what kind of thing it is"

Eg:





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.
- Each subclass inherits all the features of its superclass and adds its own unique features.
- Subclasses do not repeat the features of its superclass.

- E.g. Scrolling window and fixed window are sub classes ofwindow.

Polymorphism:

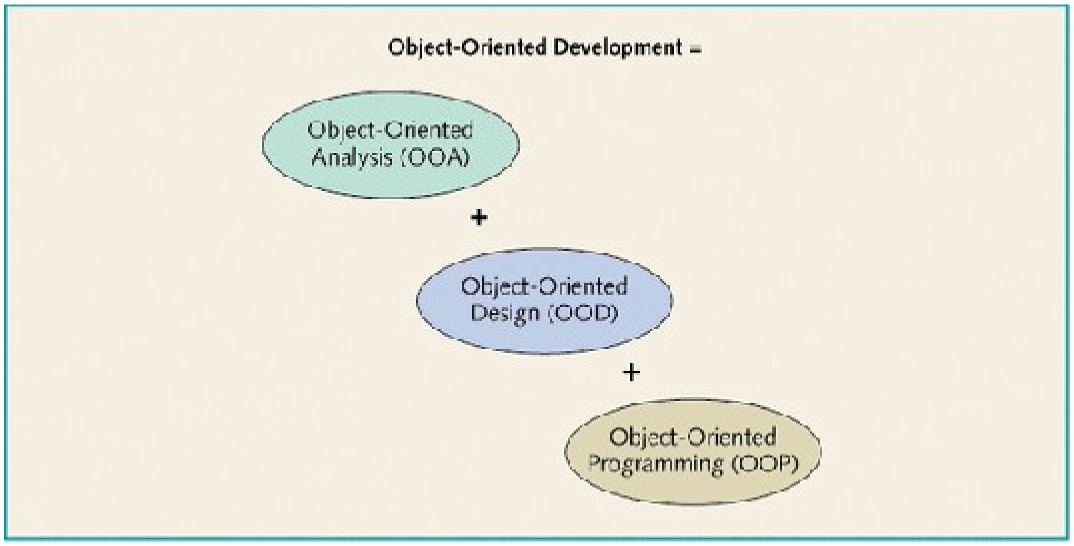
- **Polymorphism** means that the same operation may behave differently for different classes.
- For E.g. move operation behaves differently for a pawn than for the queen in a chess game.

Note: An *operation* is a procedure/transformation that an object performs or is subjected to. RightJustify, display or move are examples of operations. An implementation of an operation by a specific class is called a *method.*

In real world, an operation is simply an abstraction of analogous behavior across different kinds of objects.

In a OO programming language, the language automatically selects the correct method to implement an operation based on the name of the operation and the class of the object being operated on. (ex-chair1.move)

WHAT IS OO DEVELOPMENT? –OO development is a way of thinking about software based on abstractions that exist in real world as well in the program.


Object-Oriented Development =
Object-Oriented Analysis (OOA)
+
Object-Oriented Design (OOD)
+
Object-Oriented Programming (OOP)

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. The OO concepts and notation used to express a design also provide useful documentation.

<u>**Modeling Concepts, Not implementation**</u>

The real pay off comes from addressing front-end conceptual issues, rather than back end implementation details.

It's a conceptual process independent of programming languages. OO development is fundamentally a way of thinking and not a programming technique.

Its greatest benefits come from 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, as well as for programming.

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 duringdesign.

The methodology has the following stages

- **System conception:** Software development begins with business analysis or users conceiving an application and formulating tentativerequirements.
- **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 bedone.
  - 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 theuser.

✓ E.g. In case of stock brokerapplication-
✓ Domain objects may include- stock, bond, trade &commission.
✓ Application objects might control the execution of trades and present the results.

Application experts who are not programmers can understand and criticize a good model.

- **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 resourceallocations. Example – the system designer must decide the changes in the workstation screen must be fast and smooth, even when windows are moved or erased.
- **Class Design:** The class designer adds details to the analysis model in accordance with the system design strategy. The class designer elaborates both domain and application objects using the same OO concepts and notation. The focus of class design is the data structures and algorithms needed to implement eachclass.
- **Implementation:** Implementers translate the classes and relationships developed during class design into a particular programming language, database or hardware. Programming should be straightforward, because all the hard decisions should have already been made. During implementation, it is important to follow good software engineering practice.

OO concepts apply throughout the system development life cycle, from analysis through design to implementation. We do not consider testing as a distinct step. Ex- Developers must check analysis models against reality. Confining quality control to a separate step is more expensive and less effective.

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 interactions amongobjects.

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.

The three models are separate parts of the description of a complete system but are cross-linked. The class model is most fundamental, because it is necessary to describe what is changing or transforming before describing when or how it changes.

OO THEMES

Several themes pervade OO technology. Few are –

1. Abstraction

- Abstraction lets us 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.
- The ability to abstract is the most important skill required for OO development.

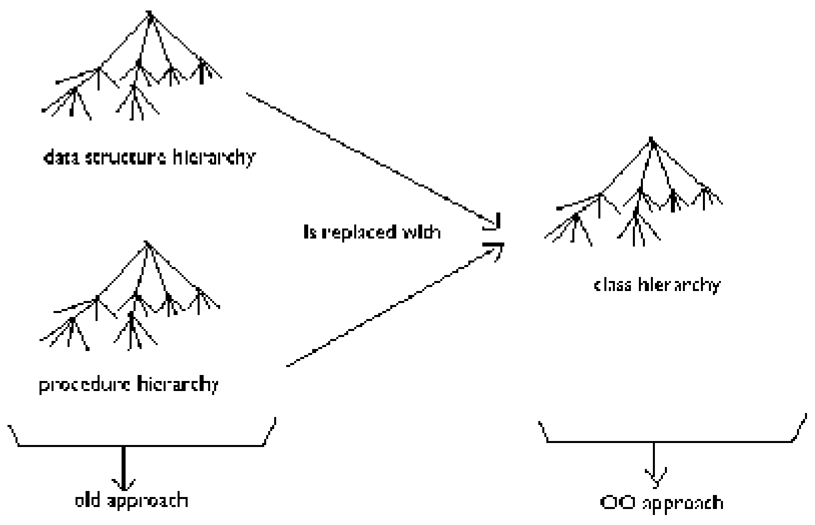
2. Encapsulation (information hiding)

- 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. We can change an object's implementation without affecting the applications that use it.

3. Combining Data and Behavior

- Caller of an operation need not consider how many implementations exist.
- Operator polymorphism shifts the burden of deciding what implementation to use from the calling code to the class hierarchy.
- Example – In a non-OO code to display the contents of a window must distinguish the type of each figure, such as polygon, circle or text and call the appropriate procedure. An OO program would simply invoke the draw operation on each figure; each object implicitly decides which procedure to use, based on its class.
- Maintenance is easier , because the calling code need not be modified when a new class is added.

➢ 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 subclasses 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. OO development provides the tools, such as abstraction, encapsulation, and inheritance, to build libraries of reusable components.

5. Emphasis on the essence of an object

- OO technology stresses what an object is, rather than how it is used. As requirements evolve, the features supplied by an object are much more stable than the ways it is used, hence software systems built on object structure are more stable in the long run.
- OO development places a greater emphasis on data structure and a lesser emphasis on procedure structure than functional-decompositionmethodologies.

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.
- The resulting system tends to be cleaner, more general and more robust than it would be if the emphasis were only on the use of data and operations.

Evidence for Usefulness of OO Development

- OO development began with internal applications at the General Electric Research and Development Center.
- OO techniques were used to develop compilers, graphics, user interfaces, databases, an OO language, CAD systems, simulations, metamodels, control systems and other applications.

- OO models to document programs that are ill-structured and difficult to understand.
- Since the mid 1990's ,the practice of OO technology expanded beyond General Electric to other companies.
- The annual OOPSLA (Object-Oriented Programming Systems, Languages, and Applications), ECOOP (European Conference on Object-Oriented Programming) and TOOLS (Technology of Object-Oriented Languages and Systems) conferences are important forums for disseminating new OO ideas and application results.
- The conference proceedings describe many applications that have been benefited from an OO approach.
- Articles on OO systems have also appeared in major publications, such as IEEE Computer and Communications of the ACM.

OO Modeling History

- The work at GE R& D led to the development of the Object Modeling Technique (OMT-previous edition of this book in 1991). The popularity of OO modeling led to a new problem- many alternative notations. The notations expressed similar ideas but had different symbols, confusing developers and making communication difficult.
- As a result, the software community began to focus on consolidating the various notations.
- In 1994 Jim Rumbaugh joined Rational and began working with Grady Booch on unifying the OMT and Booch notations.
- In 1995, Ivar Jacobson also joined Rational and added Objectory to the unification work.
- In 1996, the Object Management Group (OMG) issued a request for proposals for a standard OO modeling notation.
- Rational led the final proposal team, with Booch, Rumbaugh, and Jacobson deeply involved.
- The OMG unanimously accepted the resulting Unified Modeling language (UML) as a standard in November 1997.
- The participating companies transferred UML rights to the OMG, which owns the trademark and specification for UML and controls its future development.
- The UML notation was highly successful and replaced other notations in many publications.UML is the clearly accepted OO notation.
- In 2001 OMG members started work on a revision to add features missing from the initial specification and to fix problems in UML 1.In this we will study UML 2 which was approved in 2004. OMG web site – www.omg.org.

Chapter – 2 MODELLING AS A DESIGN TECHNIQUE

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. Examples: Architectural model to show customers, blueprints of machine parts. 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. Recent advances in computation permit the simulation of physical structures without the need to build physical models.

- **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. They can modify awkward transitions, dangling ends, and unnecessary segments before detailed writing begins.
- **Reduction of complexity:** The main reason for modeling is to deal with systems that are too complex to understand directly. Models reduce complexity to 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. A good captures the crucial aspects of a problem and omits the others. A model that contains extraneous detail unnecessarily limits the choice of design decisions and diverts the attention from the real issues.

THE THREE MODELS

We find it useful to model a system from three related but different viewpoints, each capturing important aspects of the system, but all required for a complete description.

- The Class Model
- The State Model
- The Interaction Model

A typical software procedure incorporates all three aspects: It uses data structures (class model), it sequences operations in time (state model), and it passes data and control among objects (interaction model). Each model contains references to entities in other models. Example – The class model attaches operations to classes, while the state and interaction models elaborate the operations.

The word **Model** has two dimensions – a View of a system (class model, state model or interaction model) and a stage of development (analysis, design, or implementation).

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

- It describes the structure of objects in a system- their identity, their relationships to other objects, their attributes, and their operations.
- The class model provides context for the state and interaction models.
- The goal in constructing class model is to capture those concepts from the real world that are important to anapplication.
- Class diagrams express the class model. Generalization lets classes share structure and behavior and associations relate the classes. Classes define the attribute values carried by each object and the operations that each object performs or undergoes.

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.
- The state model captures **control,** the aspect of a system that describes the

sequences of operations that occur.

- State diagram express the state model. Each state diagram shows the state and event sequences permitted in a system for one class of objects
- Each state diagram shows the state and event sequences permitted in a system for one class ofobjects.
- 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 asystem.

- Interaction model describes interactions between objects – how individual objects collaborate to achieve the behavior of the system as awhole.
- The state and interaction models describe different aspects of behavior, and we need both to describe behaviorfully.
- 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 the time sequence of their interactions. Activity diagrams show the flow of control among the processing steps of a computation.

Relationship Among the Models

Each model describes one aspect of the system but contains references to the other models. The class model describes data structure on which the state and interaction models operate. The operations in class model correspond to events and actions. The state model describes the control structure of objects. It shows decisions that depend on object values and state. The Interaction model focuses on the exchanges between the objects and provides a holistic overview of the operation of a system.

Chapter – 3 CLASS MODELLING

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.

Objects often appear as proper nouns or specific references in problem descriptions.

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.

Classes often appear as common nouns and noun phrases in problem descriptions.

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

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.

Each object knows its class. Most OO programming languages can determine an objects' class at run time. An object' class is an implicit property of the object.

Class diagrams

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

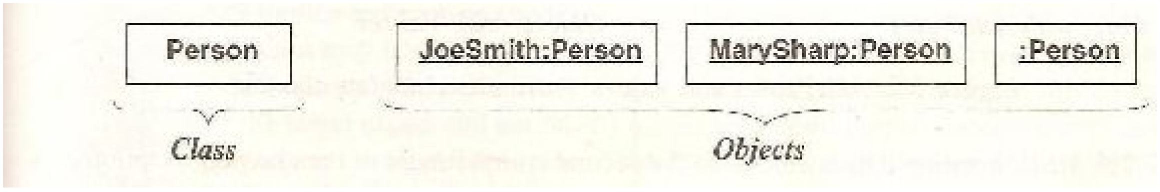
Note: An object diagram shows individual objects and their relationships.

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.
- The object name and class name are both underlined.
- Objects are modeled using box with object name followed by colon followed by class name.
- Use boldface to list the object name and 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), separating 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:

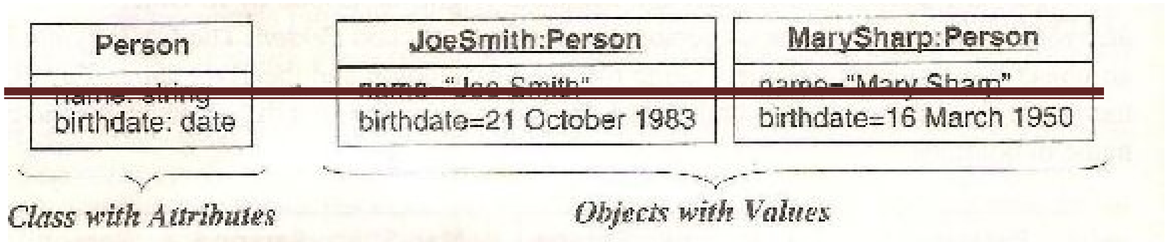
Object is to class as value is to attribute.

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

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

Different objects may have the same or different values for a given attribute. Each attribute is unique within a class. Thus class **Person** and class **Car** may each have an attribute called weight.

Fig shows modeling notation



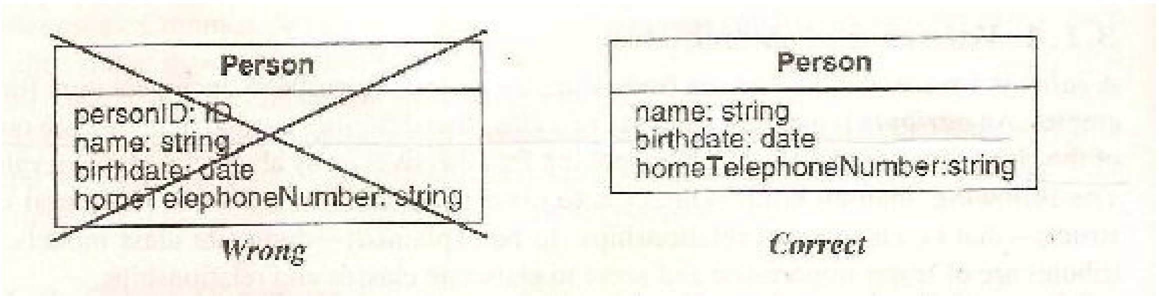
Conventions used (UML):

- 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.



Operations and Methods

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

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

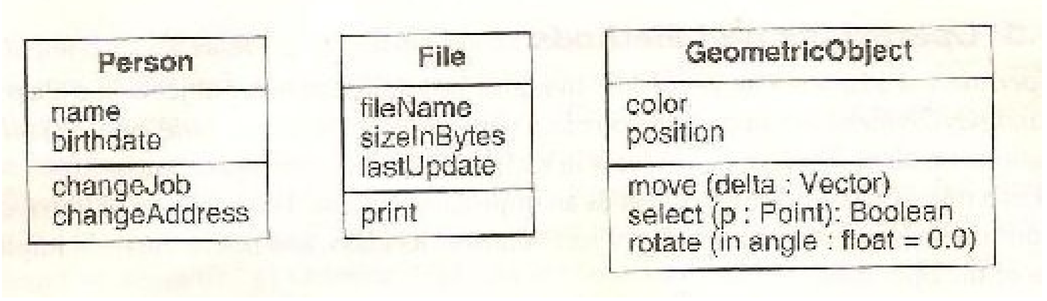
The Same operation may apply to many different classes. Such an operation is **polymorphic.**

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

In class File, may have an operation *print* - We could implement different methods to print ASCII files, print binary files etc. All these methods logically performs the same task – printing a file; thus we may refer to them by the generic operation print. A different piece of code implement each method.

When an operation has methods on several classes, it is important that the methods all have the same signature – the number and types of arguments and the type of result value. Example – *print* should not have a *filename* as an argument for one method and *filePointer* for another. The behavior of all methods for an operation should have a consistent intent.

Fig shows modeling notation.



The class Person has attributes name and birthdate and operations changeJob and changeAddress. Name, birthdate, changeJob and changeAddress are features of Person. Feature is a generic word for either an attribute or operation.

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.
- An empty argument list in parenthesis shows explicitly that there are no arguments.

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

Summary of Notation for classes

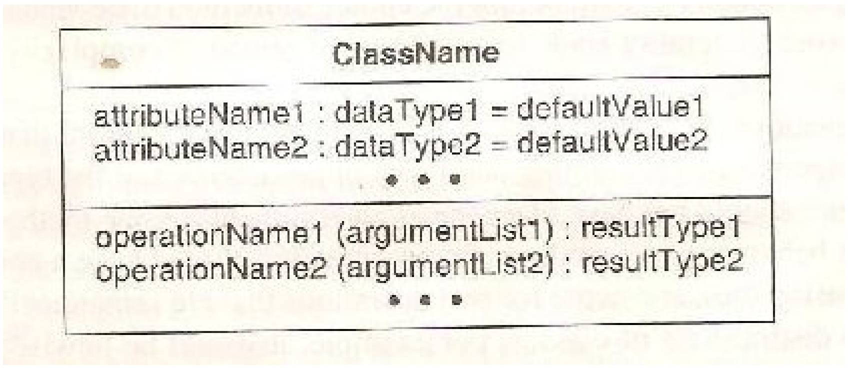

ClassName
attributeName1 : dataType1 = defaultValue1
attributeName2 : dataType2 = defaultValue2
• • •
operationName1 (argumentList1) : resultType1
operationName2 (argumentList2) : resultType2
• • •

Fig: Summary of modeling notation for classes

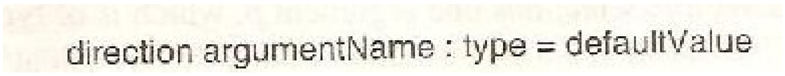
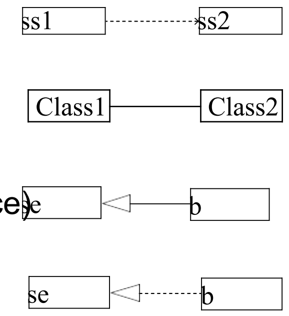


Fig: Notation for an argument of an operation

This figure shows that each argument may have a direction, name, type and default value. The **direction** indicates whether an argument is an input (in), output (out), or an input argument that can be modified(inout). A colon precedes the type. An equal sign precedes the default value. The default value is used if no argument is supplied for the argument.

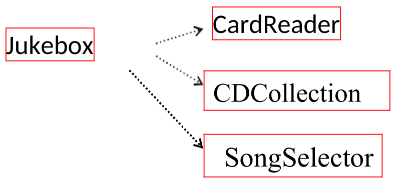
Class Digarms: Relationships

- Classescanrelatedtoeachotherthrough differentrelationships:
  - Dependency
  - Association(delegation)
  - Generalization(inheritance)
  - Realization(interfaces)



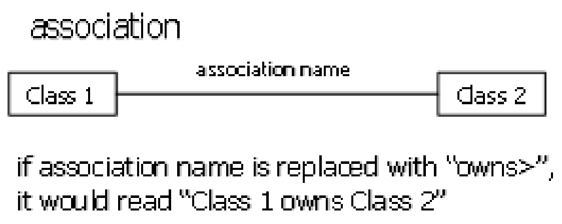
1) Dependency: A Uses Relationship

- Dependencies
  - occurs when one object depends onanother
  - if you change one object's interface, you needtochangethedependentobject
  - 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


association
association name
Class 1
Class 2
if association name is replaced with "owns>",
it would read "Class 1 owns Class 2"

LINK AND ASSOCIATION CONCEPTS

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. Most links relate two objects, but some links relate three or more objects.

E.g. JoeSmith *WorksFor* Simplex Company.

Mathematically, we define a link as a tuple – that is, a list of objects. We will be discussing only binary associations.

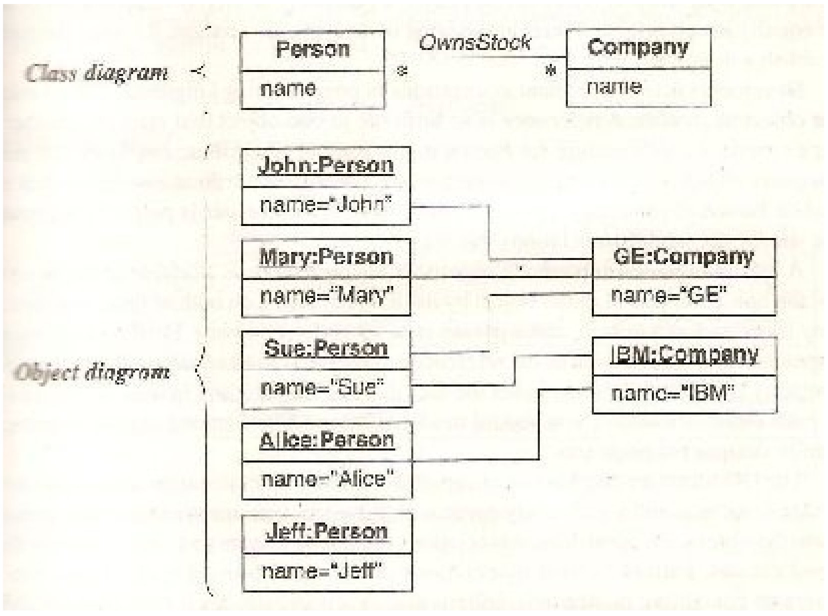
A link is an instance of an **association**.

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).

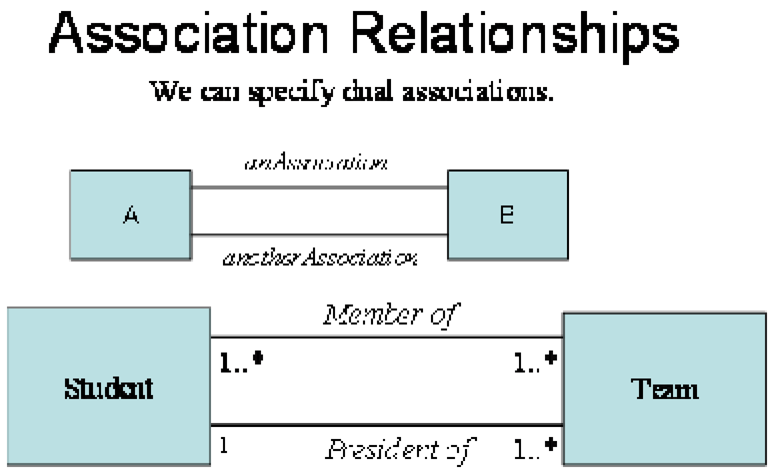


Conventions used (UML):

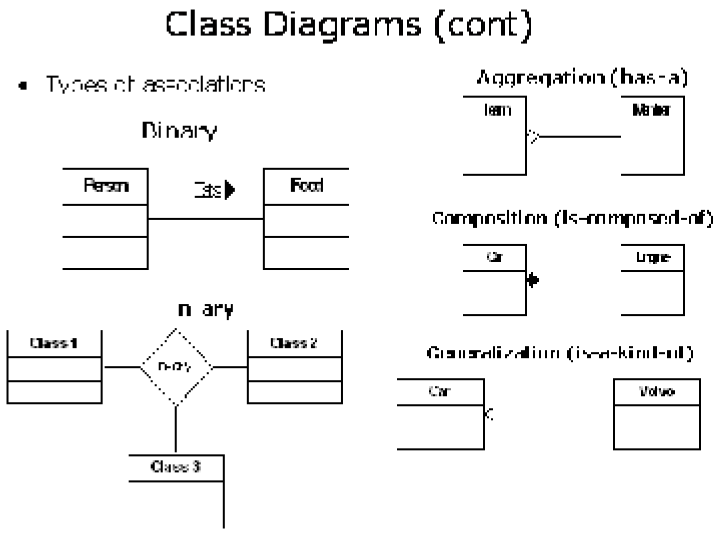
- Link is a line between objects; a line may consist of several line segments.
- If the link has the name, it is underlined.
- Association connects related classes and is also denoted by a line.
- Show link and association names in italics.

Note: In the class diagram, a person may own stock in zero or more companies; a company may have multiple persons owning its stock. The object diagram shows some examples. John, Mary and Sue own stock in the GE company.

- Association name is optional, if the model is unambiguous. Ambiguity arises when a model has multiple associations among same classes (*Person works for company and person owns stock in company)*.
- 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. For example, a data structure for Person might contain an

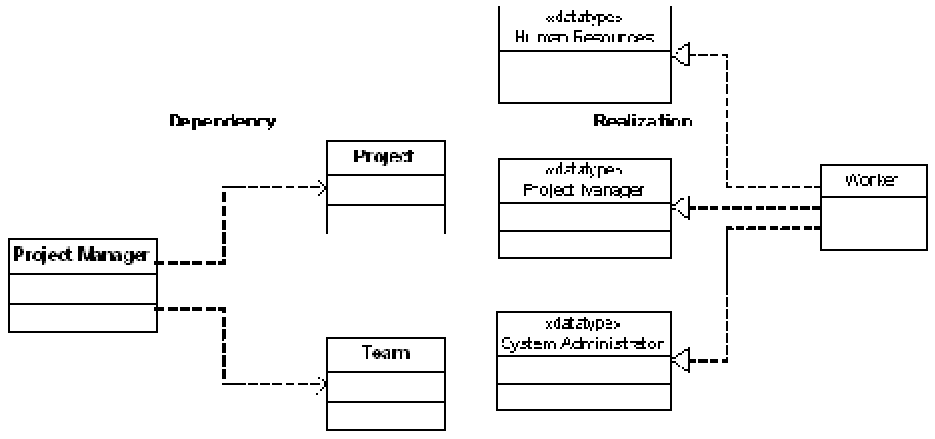


attribute employer that refers to a Company object, and a Company object might an



attribute employees that refers to a set of Person objects.

Class Diagrams (cont)



The source class depends on (uses) the target class

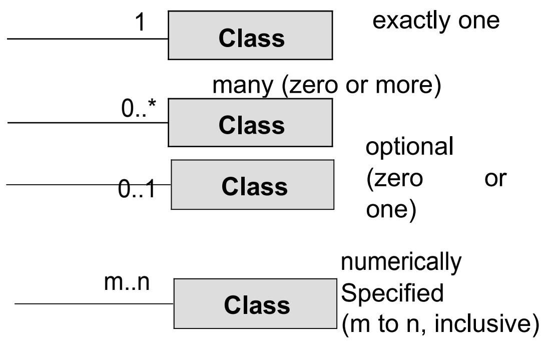
Class supports all operations of target class but not all attributes or associations.

Multiplicity

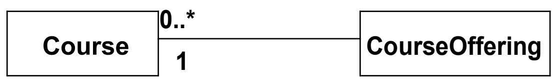
Multiplicity specifies the number of instances of one class that may relate to a single instance of an associated class. Multiplicity constrains the number of related objects.

UML conventions:

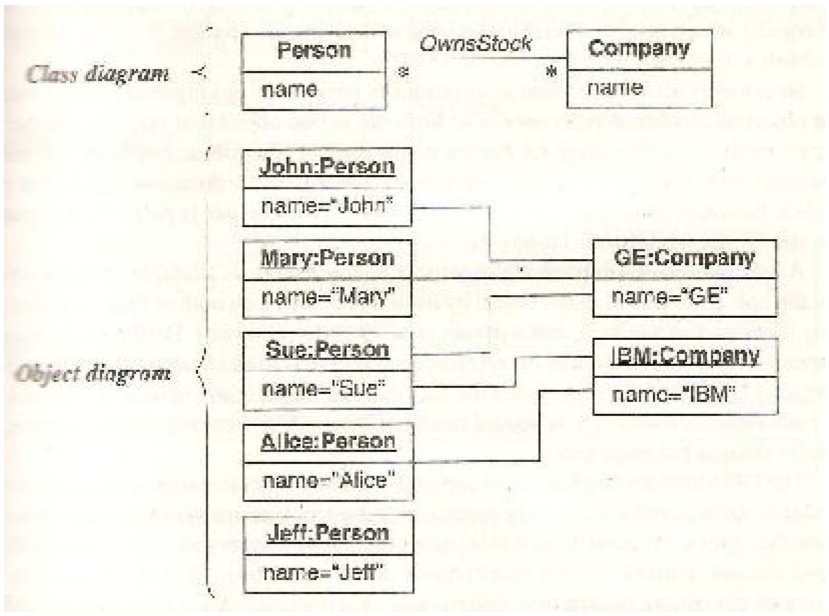
- UML diagrams explicitly lists multiplicity at the ends of association lines. UML specifies multiplicity with an interval, such as "1" (exactly one).
  "1..*"(one or more).
  "3..5"(three to five, inclusive).
  " * " (many, i.e zero or more).
- notations



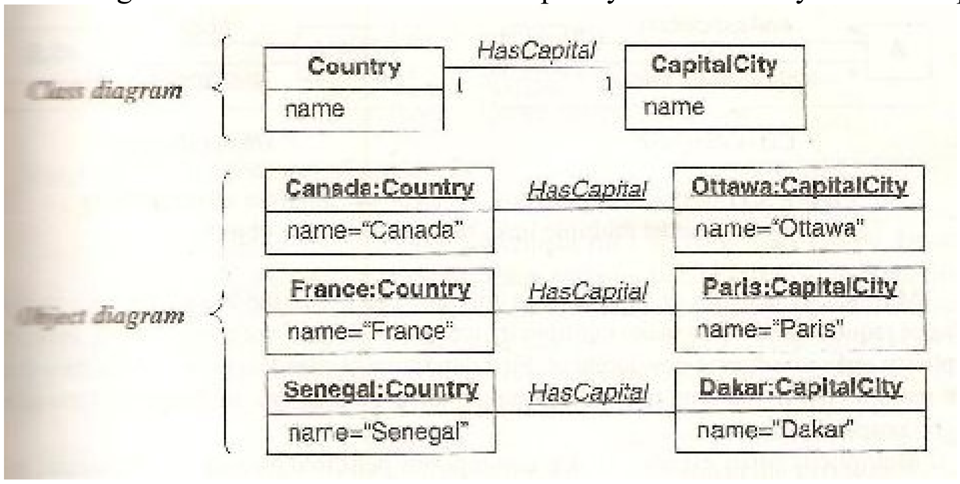
Example:



Previous figure illustrates many-to-many multiplicity. A person may own stock in many countries. A company can have multiple persons holding the stock.

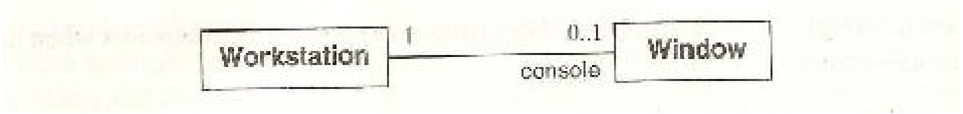


Below figure illustrates: one-to-one multiplicity. Each country has one capital city. A capital city



administers one country.

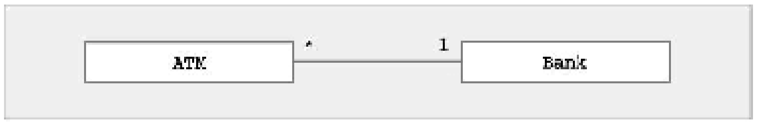
Below figure illustrates zero-or-one multiplicity. A workstation may have one of its windows designated as the console to receive general error messages. It is possible, however, that no console window exists.



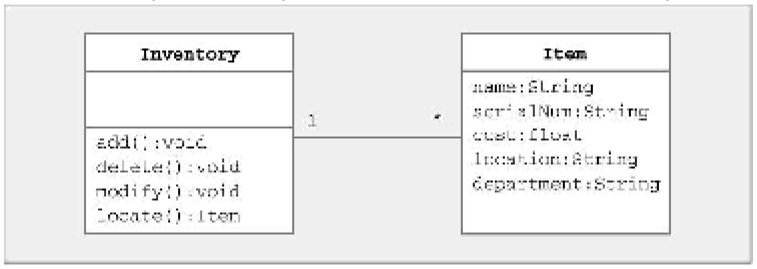
Note 1: Association vs Link.

Multiplicity of Associations

- Many-to-one
  - BankhasmanyATMs,ATMknowsonly1bank

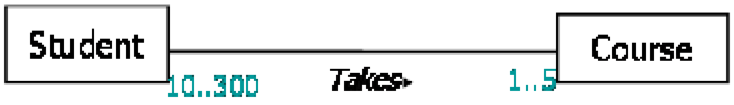


- One-to-many
  - Inventoryhasmanyitems,itemsknow1inventory



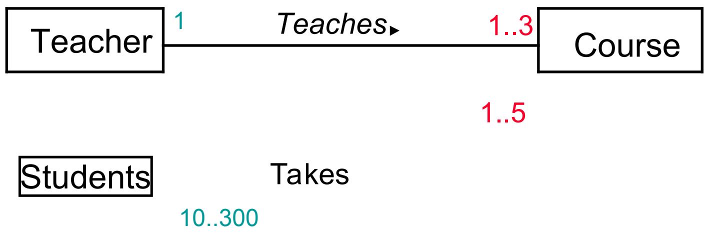
Association - Multiplicity

- 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.



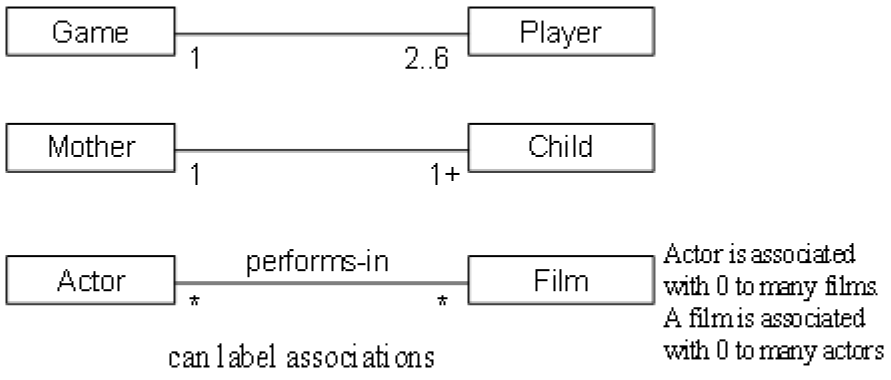
Association – Multiplicity

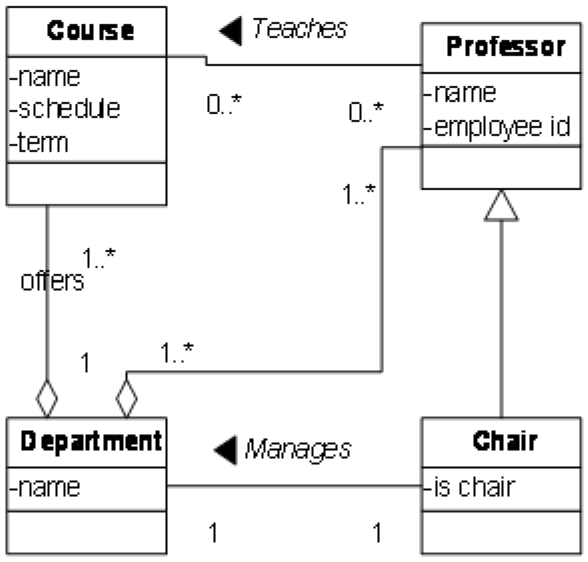
- A teacher teaches 1 to 3 courses(subjects)
- Each course is taught by only oneteacher.
- A student can take between 1 to 5courses.
- A course can have 10 to 300students.



Multiplicity

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


Game
1
2..6
Player
Mother
1
1+
Child
Actor
performs-in
*
*
Film
Actor is associated with 0 to many films
A film is associated with 0 to many actors
can label associations



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..*

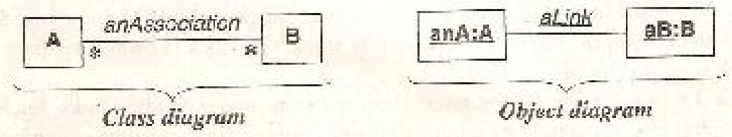


Figure 3.10 Association vs. link. A pair of objects can be instantiated at most once per association (except for bags and sequences).

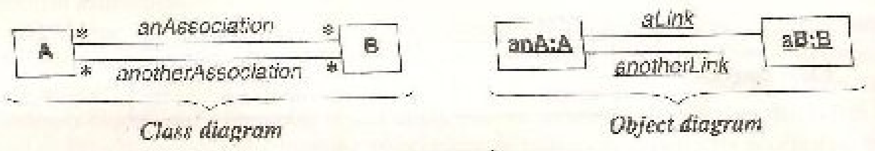


Figure 3.11 Association vs. link. You can use multiple associations to model multiple links between the same objects.

Multiplicity 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.

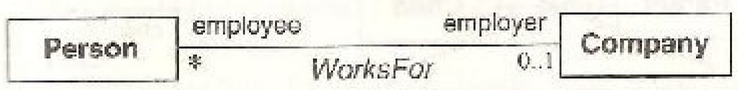
A multiplicity of "many" specifies that an object may be associated with multiple objects. However, for each association there is at most one link between a given pair of objects (except for bags and sequences).

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"

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



employee	employer
Joe Doe	Simplex
Mary Brown	Simplex
Jean Smith	United Widgets

Association end names. Each end of an association can have a name.

Association end names often appear as nouns in problem descriptions.

A person is an employee with respect to company.

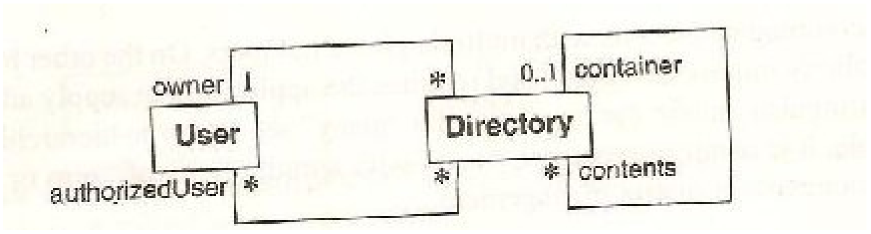
A company is an employer with respect to a person.

Association end names are optional.

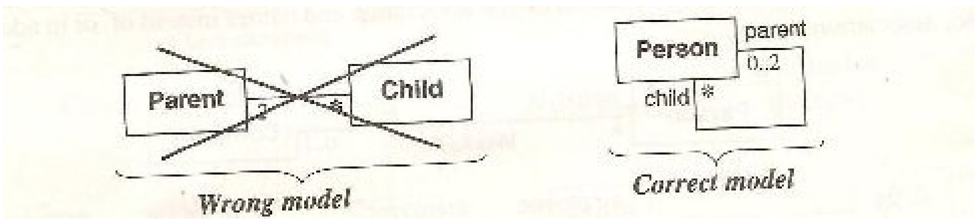
Association end names are necessary for associations between two objects of the same class. They can also distinguish multiple associations between a pair of classes.

For ex- in the figure, container and contents distinguish the two usages of Directory in the self association.

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 we may omit association end names.



Association end names lets us unify multiple references to the same class. When constructing class diagrams we should properly use association end names and not introduce a separate class for each reference as below fig shows.



Ordering

Sometimes, the objects on a "many" association end have an explicit order and we can regard them as set.

Example-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.

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

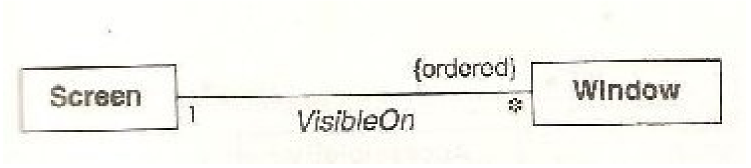


Fig: ordering sometimes occurs for "many" multiplicity

Bags and Sequences

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

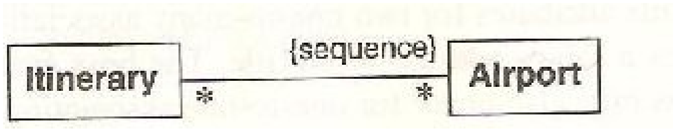


fig: an itinerary may visit multiple airports, so you should use {sequence} and not {ordered}

{ordered} and {sequence} annotations are same, except that the first disallows duplicates and the other allows them. UML1 did not permit multiple links for a pair of objects. With UML2 the intent is now clear.

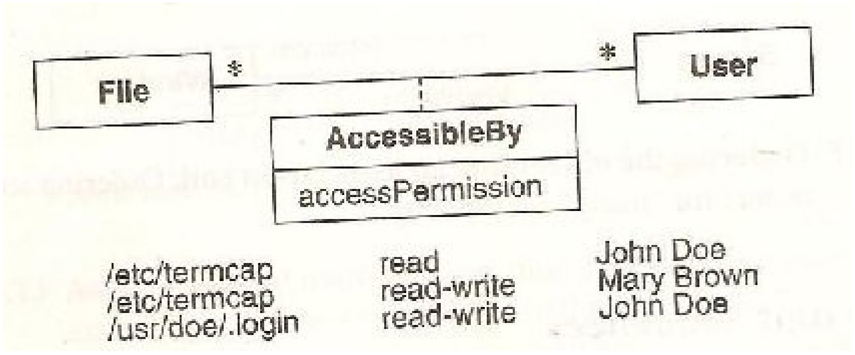
Association classes

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.

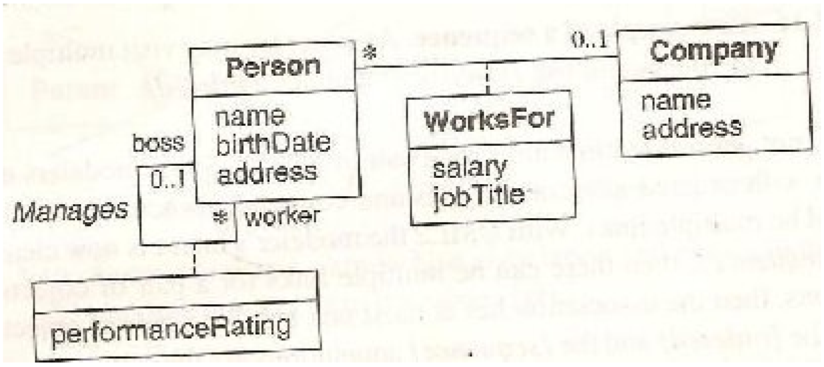
□Ex:



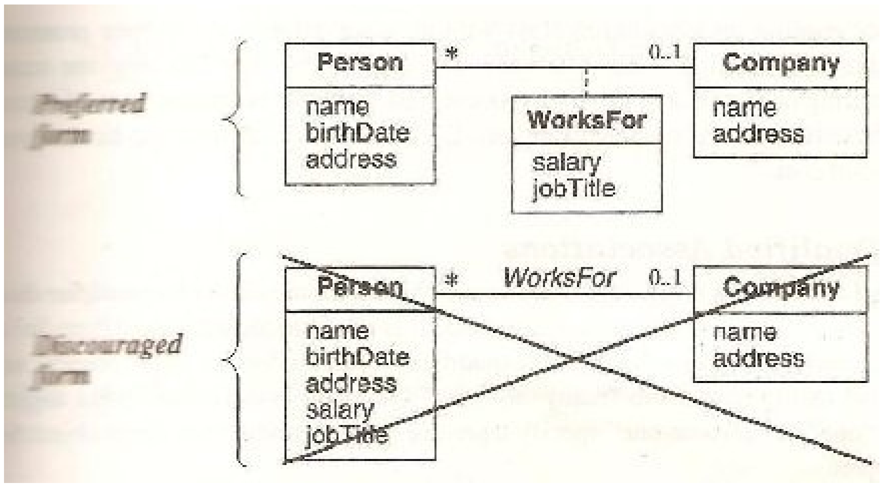
UML notation for association class is a box (class box) attached to the association by a dashed line.

Note: Many –to-many associations 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 and cannot be attached to either file or user alone without losing information.

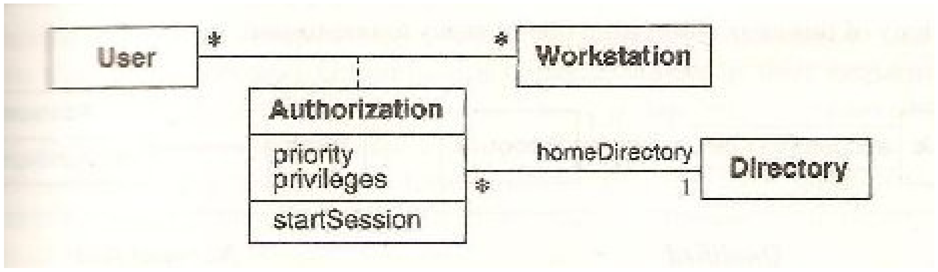
Below figure presents attributes for two one-to-many relationships. Each person working for a company receives a salary and has job title. The boss evaluates the performance of each worker. Attributes may also occur for one-to-one associations.



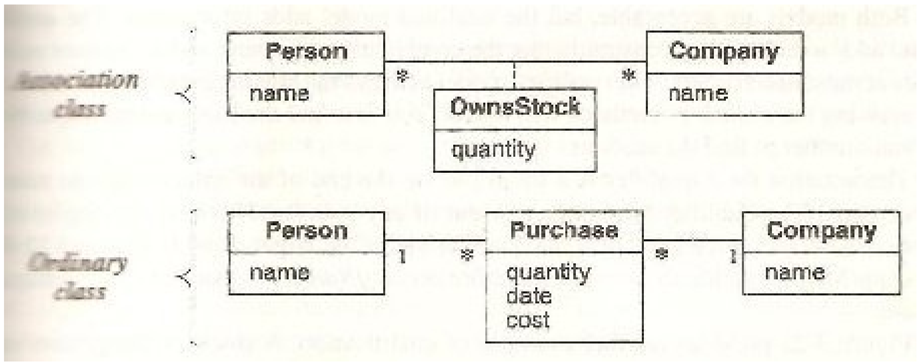
- Figure shows how it's possible to fold attributes for one-to-one and one- to-many associations into the class opposite a "one" end. This is not possible for many-to-many associations.
- As a rule, you should not fold such attributes into a class because the multiplicity of the association may change.



Proper Use of association classes. Do not fold the attributes of an association into a class



An association class participating in an association. Above figure shows an association class participating in an association. Users may be authorized on many workstations. Each authorization carries a priority and access privileges. A User has a home directory for each authorized workstation, but several workstations and users can share the same home directory.



Association class vs ordinary class.

The association class has only one occurrence for each pairing of person and Company. In constrast, there can be number of occurrences of a Purchase for each Person and Company. Each purchase is distinct and has its own quantity, date and cost.

eg:

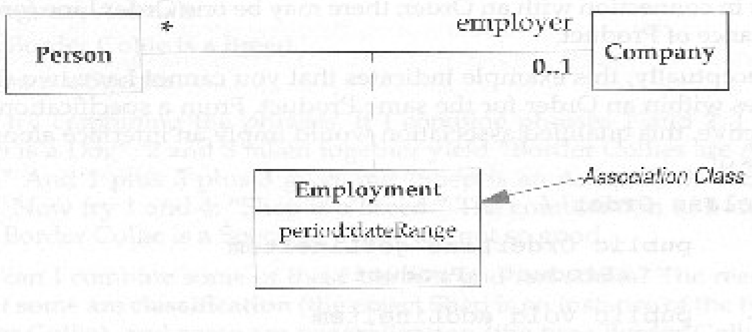
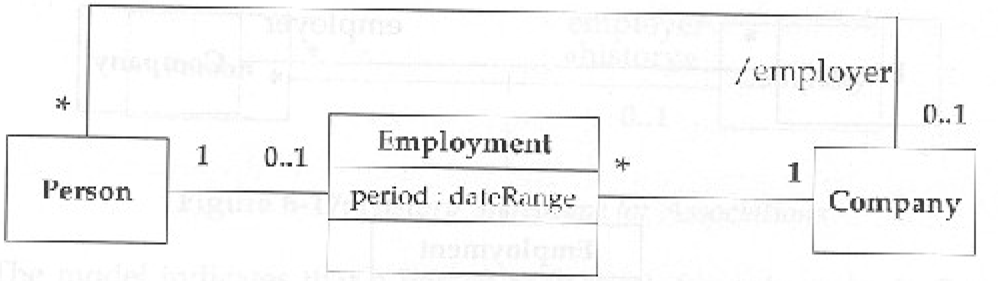


Figure 6-14: *Association Class*



Qualified associations

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

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 accountNumber is the qualifier. Qualification reduces effective multiplicity of this association from one-to-many to one-to-one.

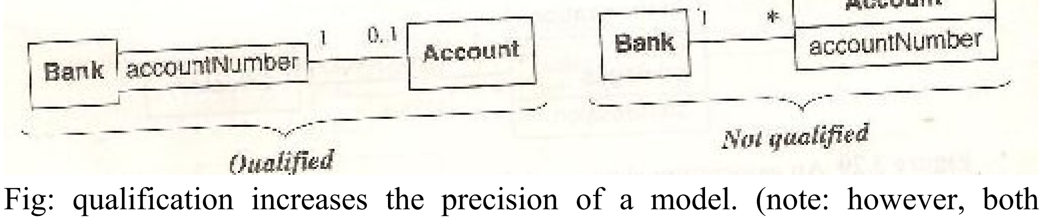
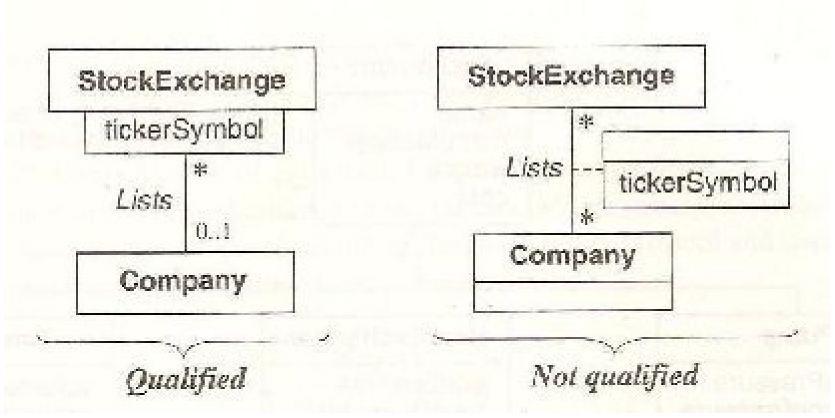
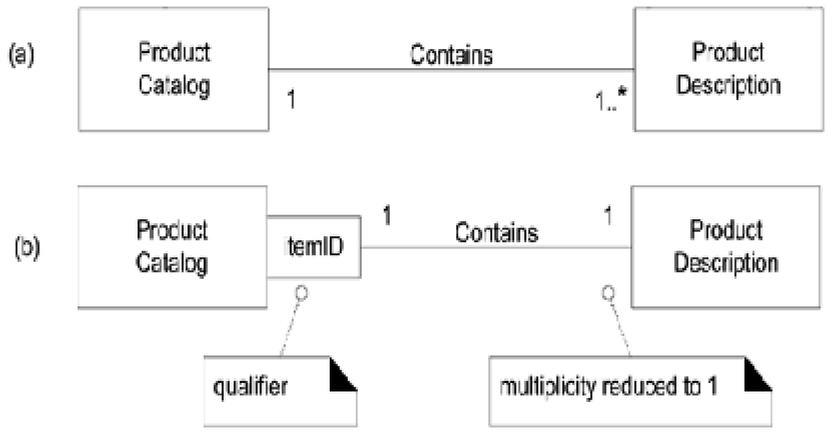


Fig: qualification increases the precision of a model. (note: however, both are acceptable)

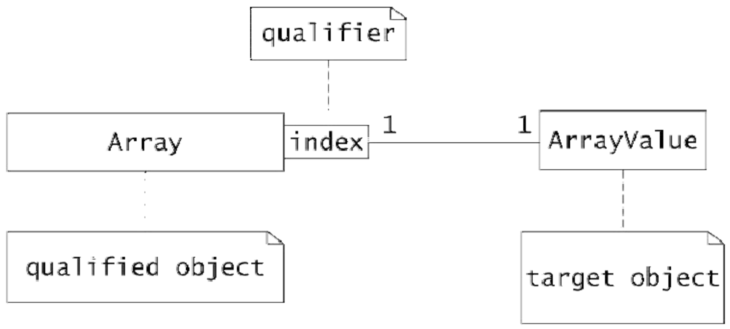
Ex 2: a stock exchange lists many companies. However, it lists only one company with a given ticker symbol. A company may be listed on many stock exchanges, possibly under different symbols.



Eg 3: Qualified Association



eg 4:



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. Generalization is sometimes called the “is-a” relationship, because each instance of a subclass is an instance of the superclass as well.

Simple generalization organizes classes into hierarchy; each subclass has a single immediate superclass. There can be **multiple levels** of generalization (subclass may have multiple immediate superclasses).

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.

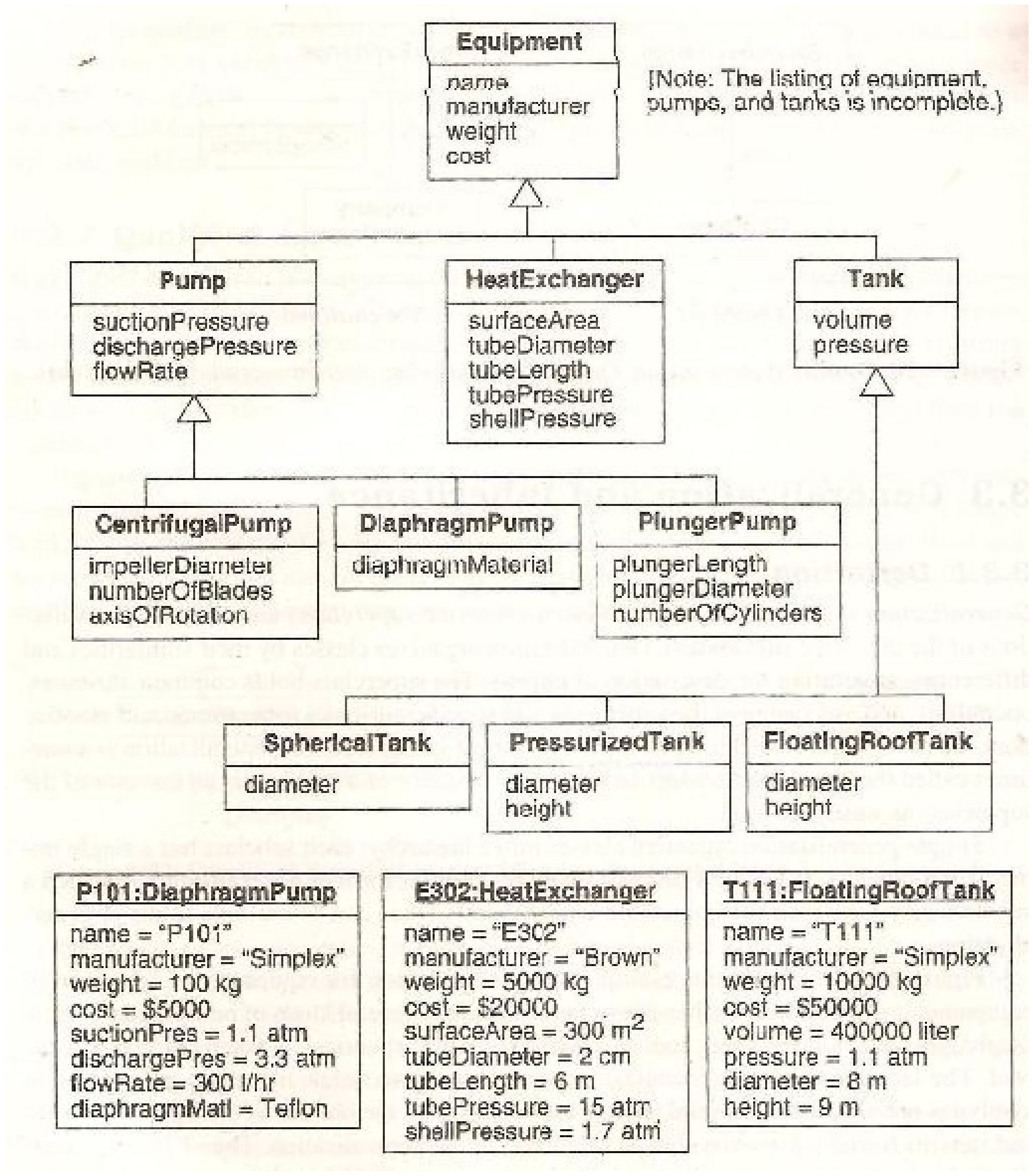
UML convention used:

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

Generalization is transitive across an arbitrary number of levels. The terms **ancestor** and **descendant** refer to generalization of classes across multiple levels. An instance of a subclass is simultaneously an instance of all its ancestor classes.

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. We should generalize only one aspect at a time. For example, the means of propulsion (wind, fuel, animal, gravity) and the operating environment(land,air,water,outer space) are two aspects for class vehicle. Generalization set values are inherently in one-to-one correspondence with the subclasses of a generalization. The generalization set name is optional.



Fig(a)**A multilevel inheritance hierarchy with instances.** Generalization organizes classes by their similarities and differences, structuring the description of objects

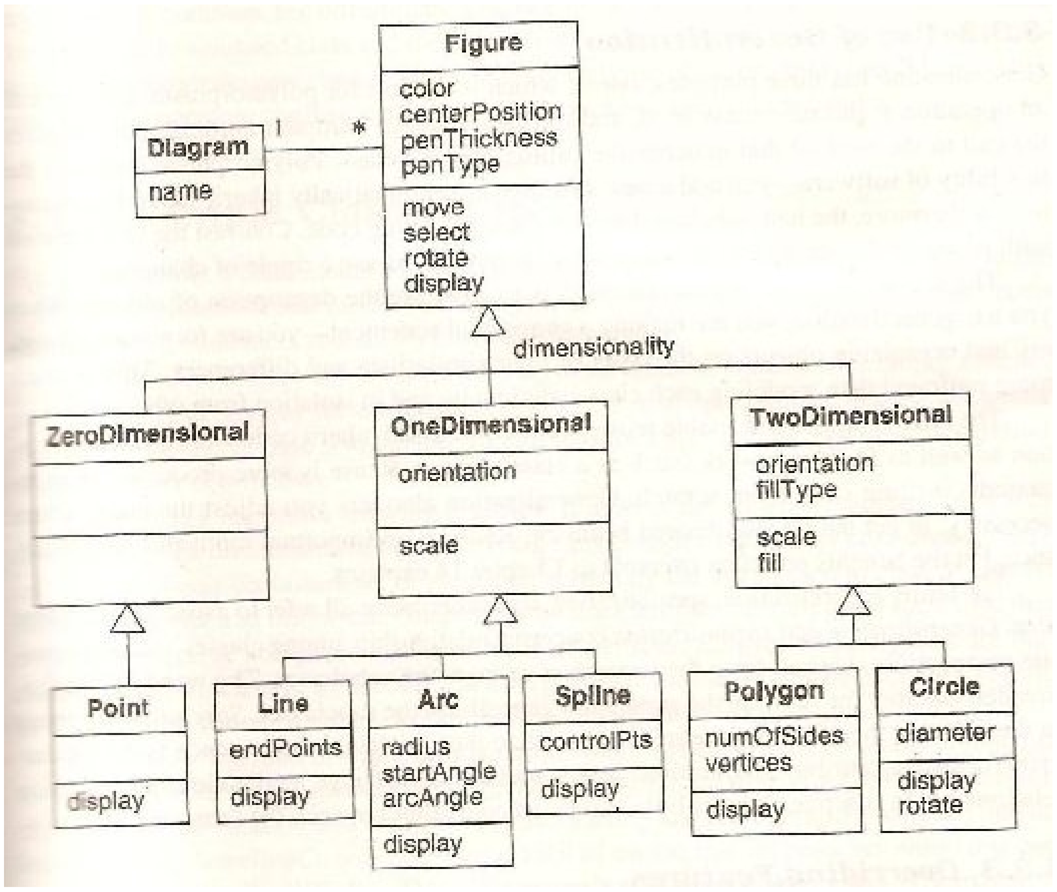


Fig (b)

'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:** We 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. Polymorphism increases the flexibility of software – We can add a new subclass and automatically inherit superclass behavior without disrupting the existing code.
2. **To structure the description of objects:** i.e to form a taxonomy and organizing objects on the basis of their similarities and differences.
3. **To enable reuse of code:** we can inherit code within our application as well as from past work. Reuse is more productive than repeatedly writing code from scratch.

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? (Reasons to have a overridden 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 must implement display, even though Figure defines it. Class Circle improves performance by overriding operation rotate to be a null operation.

Note: We may override methods and default values of attributes. We should never override the signature, or form of a feature. We should never override a feature so that it is consistent with the original inherited feature. A subclass is a special case of its super class and should be compatible with it in every respect. Otherwise, it can lead to conceptual confusion and hidden assumptions built into programs.

A SAMPLE CLASS MODEL

- Class window defines common parameters of all kinds of windows, including a rectangular boundary defined by the attributes x1,x2,y1,y2 and operations to display, undisplay a window, raise it to the top or lower it to the bottom.
- A canvas is a region of drawing graphics. It inherits the window boundary from window and adds the dimensions of the underlying canvas region defined by attribute cx1,cx2,cy1,cy2.
- A canvas contains a set of elements, shown by the association to class shape. All shape have color and line width. Shapes can be lines, ellipses or polygons, each with their own parameter.
- A polygon consists of list of vertices. Ellipse and polygon are both closed shapes, which have fill color and pattern.
- Canvas window have operation to add and delete element.
- Text window is a kind of scrolling window, which has two-dimensional scrolling off-set within its window, as specified by xoffset and yoffset as well as operation scroll to change the scroll value.
- A text window contains a string and has operations to insert and delete characters.
- Scrolling canvas is a type of canvas which supports scrolling.
- A panel contains a set of panelItem object, each identified by a unique itemName with in a given panel as shown by the qualified association.
- A panel is a predefined icon with which a user can interact on the screen.
- Panel items come in three kinds: buttons, choice and text items.
- A button has a string that appears on the screen.
- A choice item allows the user to select one of a set of predefined choices, which is a choice entry containing a string to be displayed and a value returned if entry is selected.
- There are two association between choiceItem and choiceEntry.
  a. One to many association defines the set of allowable choice.
  b. One to one association identifies the current choice. The current choice must be one of the allowable choice, so one association is a subset of the other.
- When a panel item is selected by the user, it generates an event, which is a signal that sometimes has happened together with an action to be performed.
- Each panel item has a single event, textitem have a second kind of event, which is generated when keyboard character is typed.

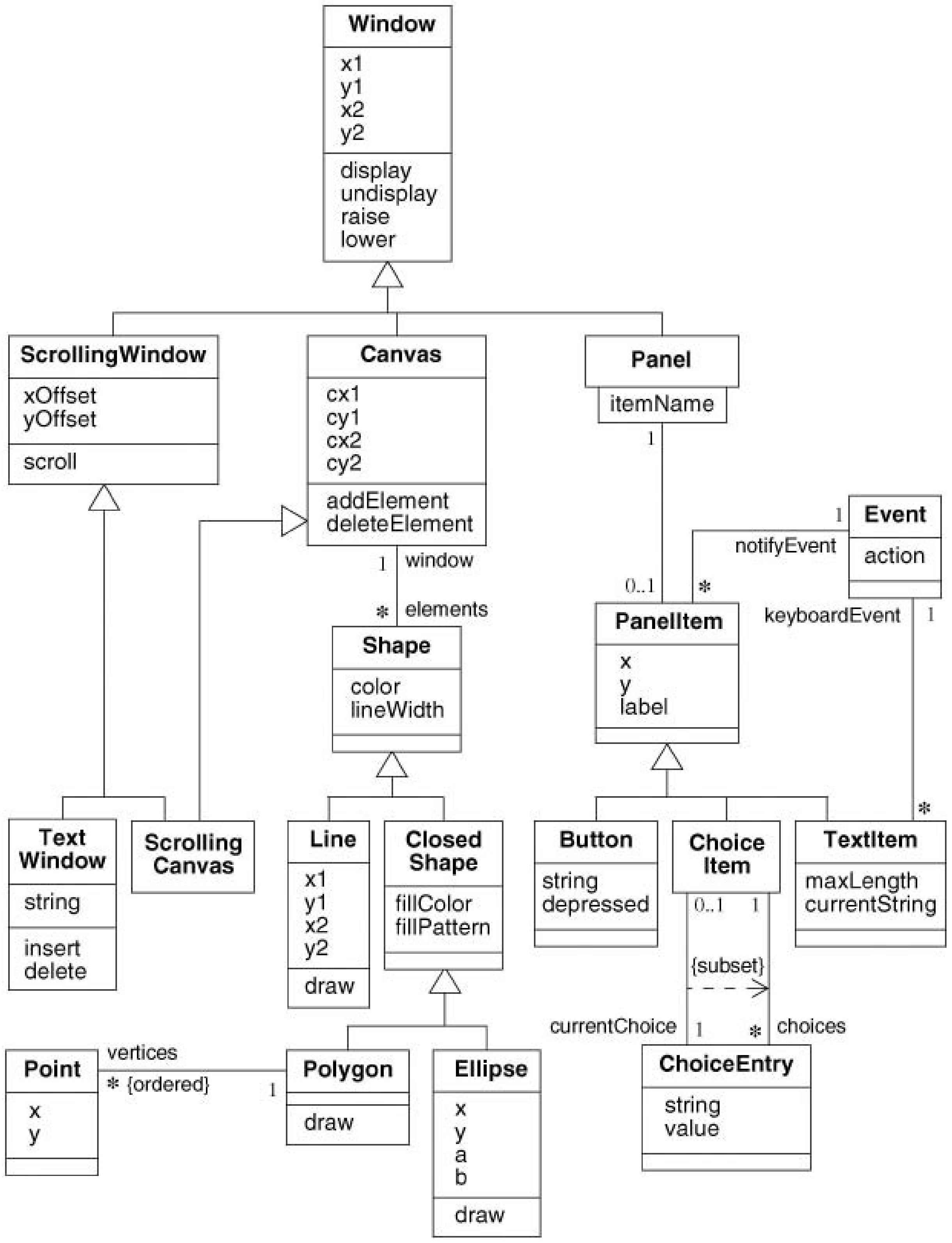
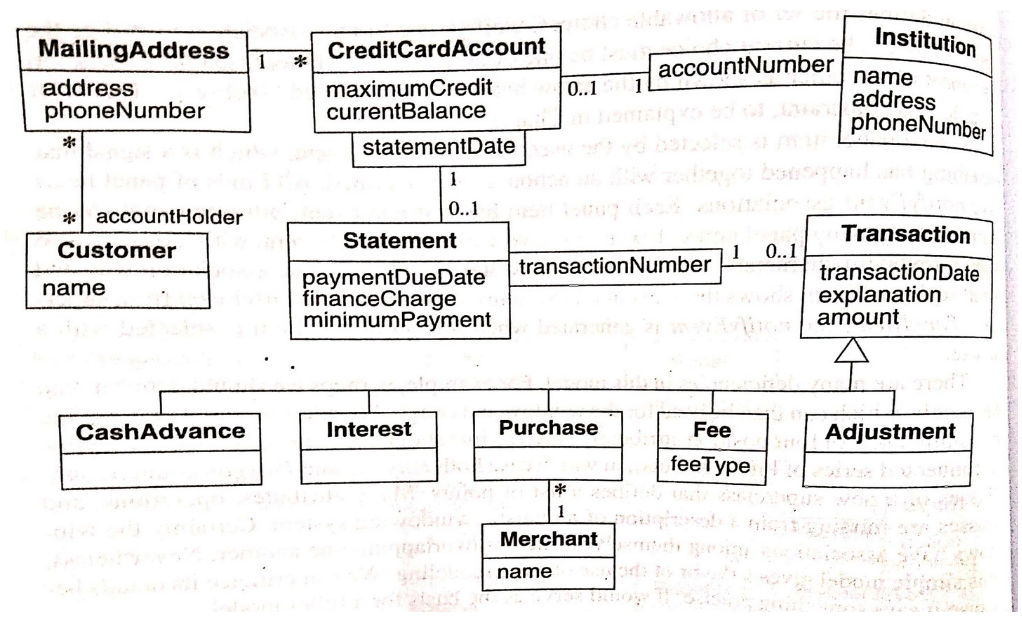


Figure 3.26 Class model of a windowing system.

NAVIGATION OF CLASS MODELS

Class models shows how they can also express the behavior of navigating among classes. are useful for more than just data structure. Furthermore, navigation exercises a class model and uncovers hidden flaws and omission, which we can then repair. We can pose variety of questions against the model.

- What transactions occurred for a credit card account within a time interval?
- What volume of transactions were handled by an institution in the last year?
- What customers patronized a merchant in the last year by any kind of credit card?
- How many credit card accounts does a customer currently have?
- What is the total maximum credit for a customer, for all accounts?



UML incorporates a language that can express these kinds of questions - the **object Constraint Language(OCL).**

OCL constructs for traversing class models (credit card questions using the OCL).

OCL can traverse the constructs in class models.

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

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

   Ex: The expression aCreditCardAccount.maximumcredit takes a CreditCardAccount object and finds the value of maximumCredit.

2. **Operations:** We can also invoke an operation for an object or acollection of objects. Syntax: source object or object collection, followed by dot and then the operation. An operation must be followed by parenthesis even if it has no arguments. The 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 may be indicated by an association end name, or class name (if there is no ambiguity).

- ➢ 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:** A qualifier lets us to make a more precise travel. 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. Alternatively, we can ignore the qualifier and traverse a qualified association as if it were a simple association. Thus the expression aCreditCardAccount. Statement finds the multiple statements for a credit card account. The multiplicity is many when the qualifier is used.

5. **Association classes:** Given a link of an association class, we can find the constituent objects. Alternatively, given a constituent object, we can find multiple links of an association class.

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

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

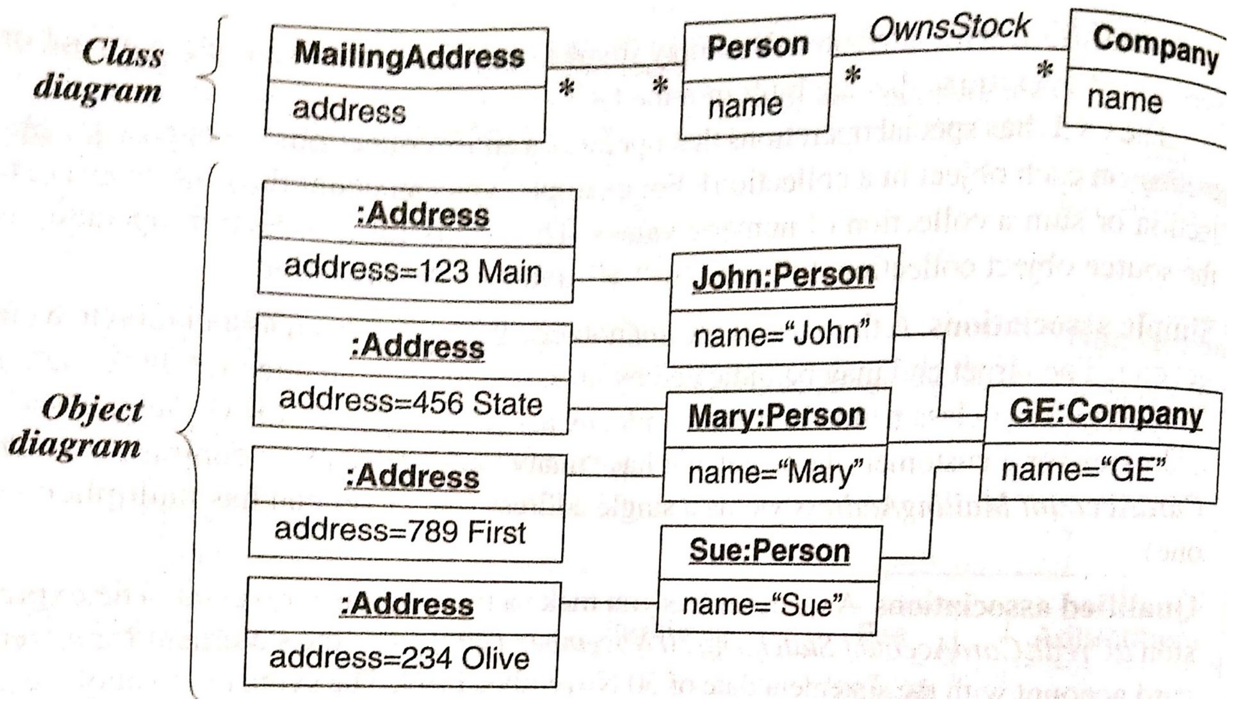
Ex: aStatement.Transaction->select(amount>$100).

It finds the transaction for a statement in excess of $100.

Building OCL Expressions

The real power of the OCL comes from combining primitive constructs into expressions. For example, an OCL expression could chain together several association traversals. There could be several qualifiers, filters and operators as well.

With the OCL, a traversal from an object through a single association yields a singleton or a set (or a bag if the association has the annotation {bag} or {sequence}). In general, a traversal through multiple associations can yield a bag (depending on multiplicities), so we must be careful with OCL expressions. A set is a collection of elements without duplicates. A bag is a collection of elements with duplicates allowed.


Class diagram
MailingAddress
address
*
*
Person
name
OwnsStock
*
*
Company
name
Object diagram
:Address
address=123 Main
:Address
address=456 State
:Address
address=789 First
:Address
address=234 Olive
John:Person
name="John"
Mary:Person
name="Mary"
Sue:Person
name="Sue"
GE:Company
name="GE"

Examples of OCL expressions

We can use the OCL to answer the credit card questions:

Write an OCL expression for –

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

Solution:

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

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

Solution:

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

What customers patronized a merchant in the last year by any kind of creditcard?

Solution: aMerchant.Purchase->
select(aStartDate<=TransactionDate and transactionDate<= anEndDate).
Statement.CreditCardAccount.MailingAddress.Cu stomer ->asset( )

How many credit card accounts does a customer currentlyhave?

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

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

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

These kinds of questions exercise a model and uncover hidden flaws and omissions that can be repaired. For example, the query on the number of credit card account suggests that we may need to differentiate past accounts from current accounts.