Module - 2

Object Oriented Modeling and Design

35

→ 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.

→ Object oriented models are useful for understanding problems, communicating with application experts, modeling enterprises, preparing documentation, and designing programs & databases.

What is Object Orientation?

Object-Oriented (OO) means that we organize software as a collection of discrete objects that incorporate both data structure and behavior.

4 characteristics are required by an OO approach

(1) Identity
(2) Classification
(3) Inheritance
(4) Polymorphism

Identity: It means that data is quantized into discrete, distinguishable entities called objects.

→ Objects can be concrete, such as a file in a file system, or conceptual, such as a scheduling policy in a multiprocessing operating system.

→ Each object has its own inherent identity (i.e, two objects are distinct even if all their attribute values are identical.

→ Ex. Mike's bicycle, a binary tree, a monitor, a symbol table etc

Classification:

→ It means that objects with the same data structure (attributes) & behavior (operations) are grouped into a class.

Ex: Paragraph, Monitor, and ChessPiece are ex's of classes

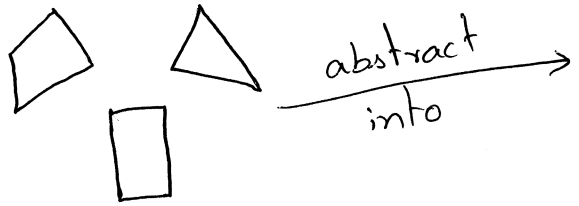
→ A class is an abstraction that describes properties important to an application.

→ Each class describes a infinite set of individual objects.

→ Each object is said to be an instance of its class.

→ An object contains an implicit reference to its own class.

Polygon objects



Polygon class
- Attributes
  - vertices
  - border color
  - fill color
- Operations
  - draw
  - erase
  - move

↳ Object has its own value for each attribute but shares the attribute names & operations with other instances of the class.

fig:- objects & classes

Inheritance (Acquires the property)

→ It is the sharing of attributes and operations (features) among classes based on a hierarchical relationship.

→ A superclass has general information that sub classes refine and elaborate.

→ Each subclass incorporates, or inherits, all the features of its superclass and adds its own unique features.

→ Subclasses need not repeat the features of the superclass.

Ex: ScrollingWindow and FixedWindow are subclasses of Window.

↳ Here both subclasses inherit the features of windows such as visible region on the screen.

↳ ScrollingWindow adds a scroll bar and an offset.

→ The ability to factor out common features of several classes into a superclass can greatly reduce repetition within designs & prgms.

It is advantage of OO technology. (Facilitates Reusability)

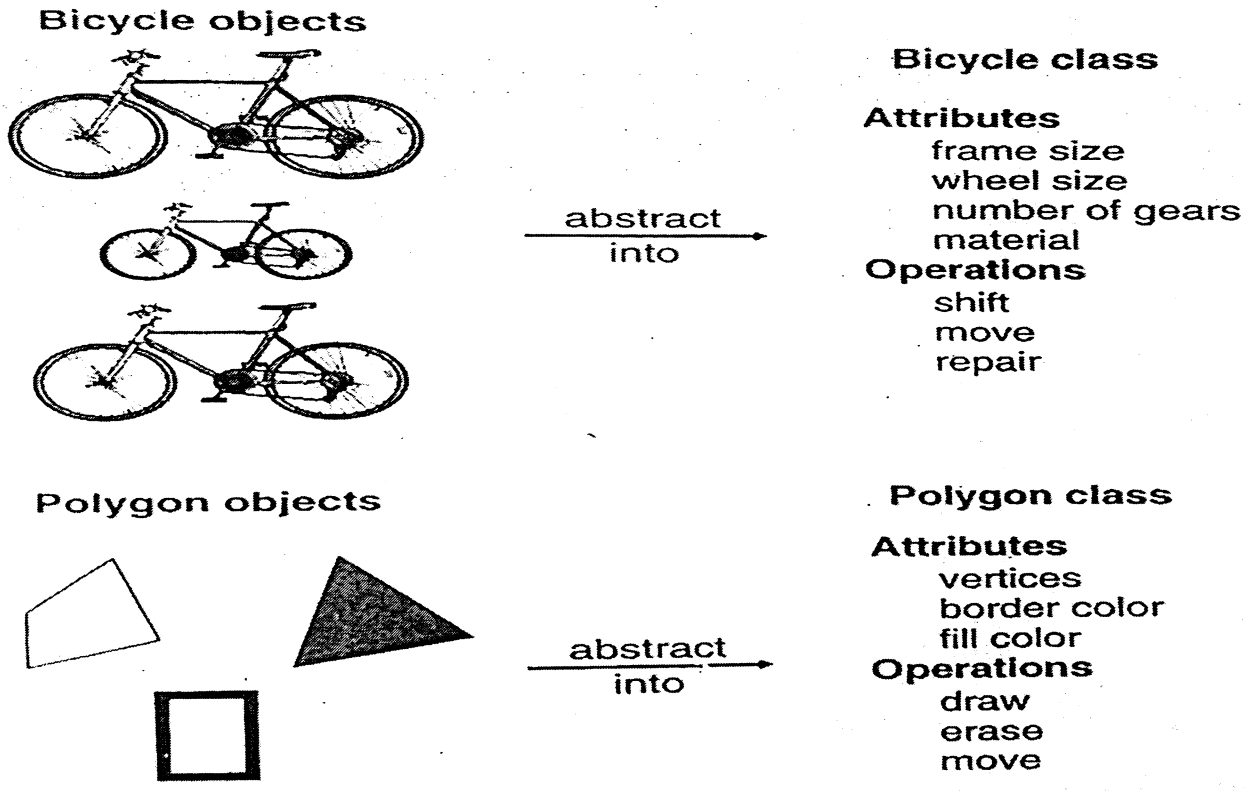


Figure 1.2 Objects and classes. Each class describes a possibly infinite set of individual objects.

Polymorphism (Ability to appear in many forms)

↳ It means that the same operation may behave differently for different classes.

↳ For Ex. the move operation behaves differently for a pawn than for the queen in chess game.

→ An operation is a procedure or transformation that an object performs or is subject to.

Ex. RightJustify, display, and move

→ Method : It is an implementation of an operation by a specific class.

→ An OO operator is polymorphic it may have more than one method implementing it, each for a different class of object.

→ In 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 object being operated on.

What is OO Development?

→ OO development is a way of thinking about s/w based on abstractions (concept) that exist in the real world as well as in program.

→ 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 & notations used to express a design also provide useful documentation.

Modeling Concepts, Not Implementation

→ An OO development approach encourages s/w developers to work and think in terms of the application throughout the s/w life cycle.

→ OO development is a conceptual process independent of a programming language until the final stages.

→ OO development is fundamentally a way of thinking & not a programming technique.

→ Its greatest benifits come from helping specifiers, developers and customers express abstract concepts clearly & communicate them to each other.

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

OO Methodology

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: S/w development begins with business analysts or users conceiving an application and formulating tentative requirements.

* Analysis: The analyst scrutinizes and rigorously restates the requirements from system conception by constructing models.

→ The analyst must work with the requestor to understand the problem, because problem statements are rarely complete or correct.

→ The analysis model is a concise, precise abstraction of what the desired system must do; not how it will be done.

→ The analysis model should not contain implementation decisions.

→ The analysis model has two parts:

* Domain model:- a description of the real world objects reflected within the system;

* Application model - a description of the parts of the application system itself that are visible to the user.

Ex: In case of stock brocker 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 team devise a high level strategy - The system architecture - for solving the application problem.

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

* 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 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 s/w engineering practice so that traceability to the design is apparent & so that the system remains flexible and extensible.

→ OO concepts apply throughout the system development life cycle from analysis through design to implementation.

→ We can carry the same classes from stage to stage without a change of notation, although they gain additional details in the later stages.

→ The same OO concepts of identity, classification, polymorphism & inheritance apply throughout development.

→ Some classes are not part of analysis but are introduced during design or implementation.

Three Models

Three kinds of models are used to describe a system from different viewpoints:

* class model - for the objects in the system and their relationships.
* State model - for the life history of objects.
* Interaction model - for the interactions among objects.

→ The class model describes the static structure of the objects in a system and their relationships.

↳ The class model defines the context for s/w development.

↳ class model contains class diagrams

↳ A class diagram is a graph whose nodes are classes and whose arcs are relationships among classes.

→ The state model describes the aspects of an object that change over time.
↳ The state model specifies and implements control with state diagrams.
↳ A state diagram is a graph whose nodes are states and whose arcs are transitions between states caused by events.
→ The interaction model describes how the objects in a system cooperate to achieve broader results.
↳ The interaction model starts with use cases that are then elaborated with sequence and activity diagrams.
↳ A use case diagram focuses on the functionality of a system - i.e, what a system does for users.
↳ A sequence diagram shows the objects that interact and the time sequence of their interactions.
↳ An activity diagram elaborates important processing steps.
→ The 3 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 spread OO technology

↳ Abstraction

Abstraction lets us focus on essential aspects of an application while ignoring details.
↳ This means focusing on what an object is and does, before deciding how to implement it.
→ Use of abstraction preserves the freedom to make decisions as long as possible by avoiding premature commitments to details.

↳ Encapsulation

Encapsulation (also information hiding) 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.

→ The ability to combine data structure and behavior in a single entity makes encapsulation cleaner and more powerful.

↳ Combining Data and Behavior

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

→ 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.

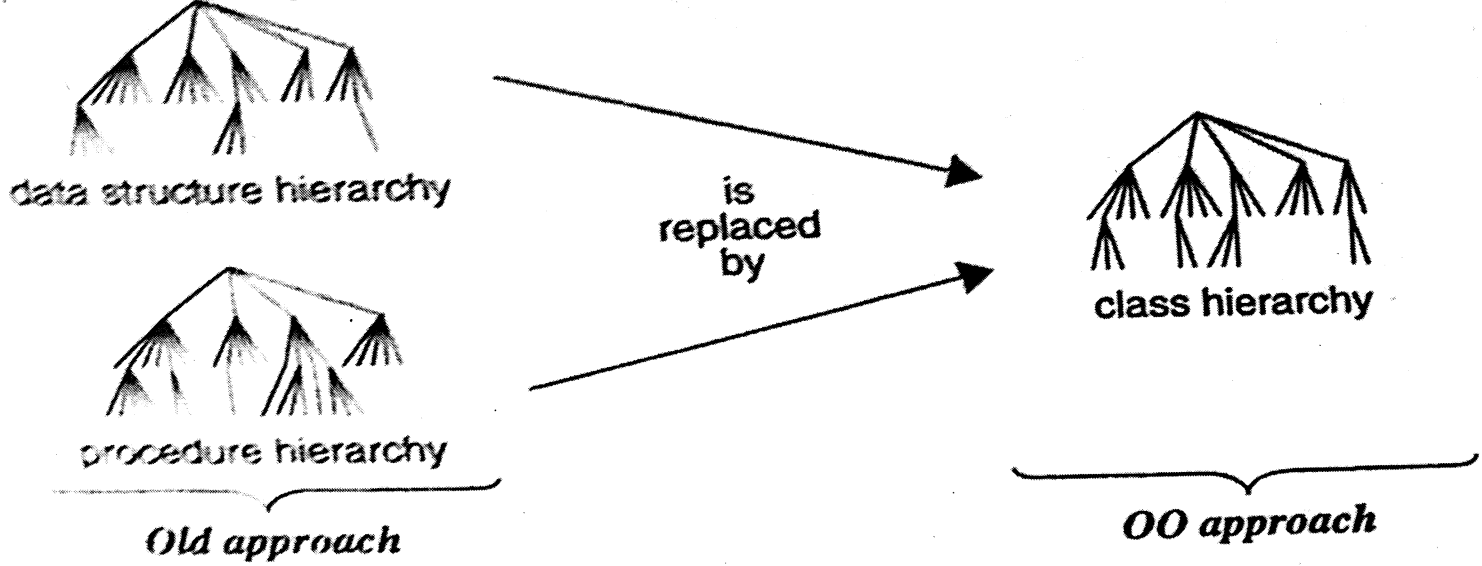


Figure 1.3 OO vs. prior approach. An OO approach has one unified

↳ Sharing

→ OO techniques promote sharing at different levels.
→ Inheritance of both data structure and behavior lets subclasses share common code. This sharing via inheritance is one of the main advantages of OO languages.
→ OO development not only lets us 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.

↳ Emphasis on the Essence of an Object

→ OO technology stresses what an object is, rather than how it is used.
→ The use of an object depend on the details of the application and often change during development.
→ OO development places a greater emphasis on data structure & a lesser emphasis on procedure structure than functional decomposition methodologies.

↳ 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.
→

Evidence for Usefulness of OO Development

→ Used OO techniques for developing compilers, graphics, user interfaces, databases, an OO language, CAD systems, simulations metamodels, control systems, and other applications.

→ Used OO models to document programs that are ill structured and difficult to understand.

OO Modeling History

→ Work at GE R&D led to the development of the Object Modeling Technique (OMT), 1991.

→ The popularity of OO modeling led to a new problem - alternative notations. The notations expressed similar ideas but had different symbols, confusing developers & making communication difficult.

→ In 1994 Jim Rumbaugh joined Rational & began working with Grady Booch on unifying the OMT & Booch notations.

→ In 1995, Ivar Jacobson also joined Rational & added Objectory to the unification work.

→ In 1996 the object Management Group (OMG) issued a request for proposals for standard OO modeling notations.

→ The OMG accepted Unified Modeling Language (UML) as standard in Nov 1997 which is developed by team Booch, Rumbaugh, & Jacobson

→ The UML was highly successful & replaced the other notations in the most publications.

→ In 2001 OMG members started work on a revision to add missing features from UML 1. Then UML 2.0 revision approved in 2004.

Modeling as a Design Technique

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

Modeling

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

Models serve several purposes:-

↳ Testing a physical entity before building it.

The medieval masons did not know modern physics, but they built scale models of the Gothic cathedrals to test the forces on the structure.

↳ Engineers test scale models of airplanes, cars, & boats in wind tunnels and water tanks to improve their dynamics.

↳ Both physical models & computer models are usually cheaper than building a complete system & enable early correction of flaws

Communication with customers:

↳ Architects and product designers build models to show their customers. Mock-ups are demonstration products that imitate some or all of the external behavior of a system.

Visualization

↳ Storyboards of movies, television 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

→ Modeling is to deal with systems that are too complex to understand directly.

↳ Models reduce complexity by separating out a small no. of important things to deal 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.

→ Many different abstractions of the same thing are possible depending on the purpose for which they are made.

→ All abstractions are incomplete & inaccurate.

→ The purpose of an abstraction is to limit the universe so we can understand.

→ A good model captures the crucial aspects of a problem and omits the others.

The Three Models

→ The class model represents the static, structural, "data" aspects of a system.

→ The state model represents the temporal, behavioral, "control" aspects of a system.

→ The interaction model represents the collaboration of individual objects, the "interaction" aspects of a system.

→ A typical s/w procedure incorporates all 3 aspects: It uses data structures (class model), it sequences operations in time (state model) & it passes data & control among objects (interaction model).

Class Model

→ The class model 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 & interaction models.

→ 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.

→ classes define the attribute values carried by each object and the operations that each object performs or undergoes.

State Model

The 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, & the organization of events and states.

→ State diagrams express the state model.

→ Each state diagram shows the state and event sequences permitted in a system for one class of objects.

→ State diagrams 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.

Interaction Model

→ It describes interactions between objects - how individual objects collaborate to achive the behavior of the system as a whole.

→ The state & interaction models describe different aspects of behavior & we need both to describe behaviour fully.

→ Use cases, sequence diagrams, & activity diagrams document the interaction model.
→ Use cases document major themes for interaction blw 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 the 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 causes actions that change object value and state.
→ The interaction model focuses on the exchanges between objects and provides a holistic overview of the operation of a system.

SWATHI · G
RNSIT

continue

Module - 2 Class Modeling

→ A class model captures the static structure of a system by characterizing the objects in the system, the relationships b/w the objects, & the attributes and operations for each class of objects.

Object and Class Concepts

→ The Objects

→ The 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 No. 7648 & top window etc.

→ Objects often appear as proper nouns or specific references in problem descriptions & discussions with users.

→ All objects have identity and are distinguishable.

→ The term identity means that objects are distinguished by their inherent existence and not by descriptive properties that they may have.

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, & semantics.

Ex: Person, company, process, and window are all classes
Each person has name & birthdate & may work at a job.
Each process has an owner, priority, & list of required resources.

→ Objects in a class have the same attributes and forms of behavior.
→ Most objects derive their individuality from differences in their attribute values and specific relationships to other objects.
→ Objects with identical attribute values and relationships are possible.
→ The objects in a class share a common semantic purpose above and beyond the requirement of common attributes and behavior.
→ The interpretation of semantics depends on the purpose of each application and is a matter of judgment.
→ Each object "knows" its class.
→ Most OO programming languages can determine an object's class at run time.
→ An object's class is an implicit property of the object.
→ We can write operations once for each class, so that all the objects in the class benefit from code reuse.

Class Diagrams : There are 2 kinds of models of structure < class diagrams / Object diagrams

→ Class diagrams provide a graphic notation for modeling classes and their relationships, thereby describing possible objects.
→ Class diagrams are useful both for abstract modeling and for designing actual programs.
→ An object diagram shows individual objects and their relationships.
→ Object diagrams are helpful for documenting test cases and discussing examples.
→ A class diagram corresponds to an infinite set of object diagrams.

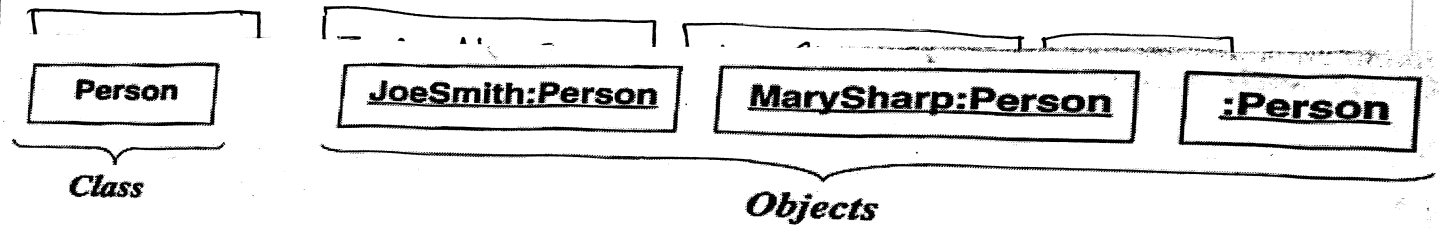


Figure 3.1 A class and objects. Objects and classes are the focus of class modeling.

→ Objects JoeSmith, MarySharp & an anonymous person are instances of class Person.

→ The UML symbol for an object is a box with an object name followed by a colon and the class name.

→ The object name and class name are both underlined.

→ Use boldface to list the object name & class name.

→ The UML symbol for a class also is a box.

→ List the class name in boldface, center the name in the box, and capitalize the first letter. Use singular nouns for the names of classes.

→ To run together multiword names (such as JoeSmith), separate the words with intervening capital letter.

Values and Attributes

→ A value is a piece of data.

→ An attribute is a named property of a class that describes a value held by each object of the class.

→ Object is to class as value is to attribute.

Name, birthdate & weight are attributes of Person objects.
color, modelYear, & weight are attributes of Car objects.

→ Each attribute has a value for each object.

Values: JoeSmith, 21October1983, 64 (of person object).

→ Each attribute name is unique within a class. Thus class Person and class Car may each have an attribute called weight.
→ An attribute should describe values, not objects.

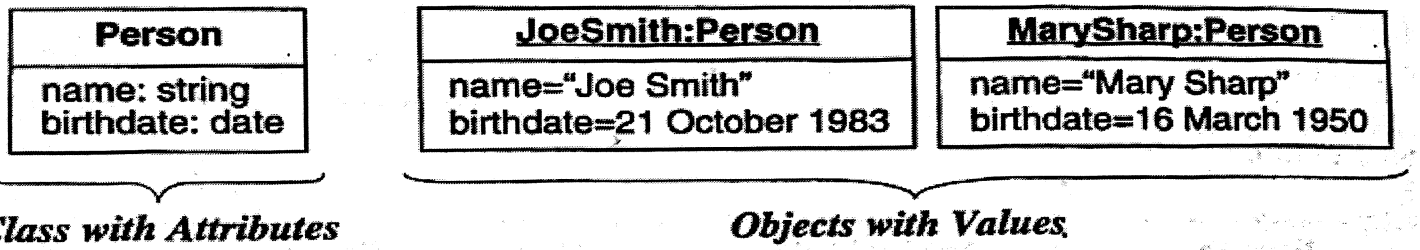


Figure 3.2 Attributes and values. Attributes elaborate classes.

→ Class Person has attributes name and birthdate. Name is string and birthdate is a date.
→ One object in class Person has the value "Joe Smith" for name and the value "21 October 1983" for birthdate.
→ Another object has the value "Mary Sharp" for name & the value "16 March 1950" for birthdate.
→ The UML notation lists attributes in the second component of the class box. Optional details (like type & default value) may follow each attribute.
→ A colon precedes the type. An equal sign precedes the default value.
→ Show the attribute name in regular face, left align the name in the box, & use a lowercase letter for the first letter.
→ Ily we may also include attribute values in the second compartment of object boxes with same conventions.
→ Do not list object identifiers; they are implicit in a class model.

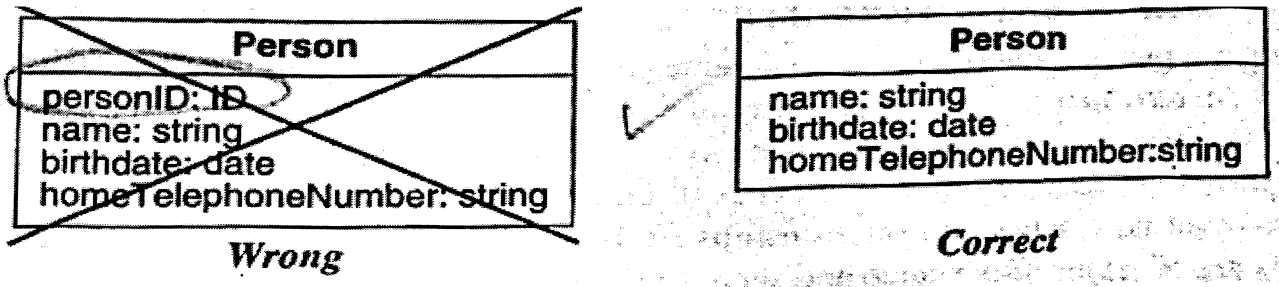


Figure 3.3 Object identifiers. Do not list object identifiers; they are implicit in models.

Operations and Methods

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

Ex: Open, close, hide and redisplay are operations on class Window.

→ All objects in a class share the same operations.

→ Each operation has a target object as an implicit argument.

→ The same operation may apply to many different classes. Such operation is polymorphic. ie, the same operation takes on different forms in different classes.

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

Ex: The class File may have an operation print, we could implement different methods to print ASCII files, print binary files & print digital picture files. All these methods perform same task printing a file. we can refer them by generic operation print.

→ An operation may have arguments in addition to its target object. Such arguments may be placeholders for values, or for other objects.

The class Person has attributes name & birthdate & operations changeJob & changeAddress. These are features of Person.

→ Feature is a generic word for either an attribute or operation.

→ My File has print operation.

→ GeometricObject has move, select and rotate operations

→ Move has argument delta, which is a Vector, select has argument p, which is of type Point & returns a Boolean. rotate has argument angle, which is an i/p of type float with a default value of 0.0.

→ The UML notation is to list operations in the 3rd compartment of the class box.

→ To list the operation name in regular face, left-align the name in the box, and use a lowercase letter for the first letter.

→ Optional details, such as an argument list and result type, may follow each operation name.

→ Parentheses enclose an argument list; Commas separate the arguments. A colon precedes the result type,

→ An empty argument list in parantheses shows explicitly that there are no arguments;

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

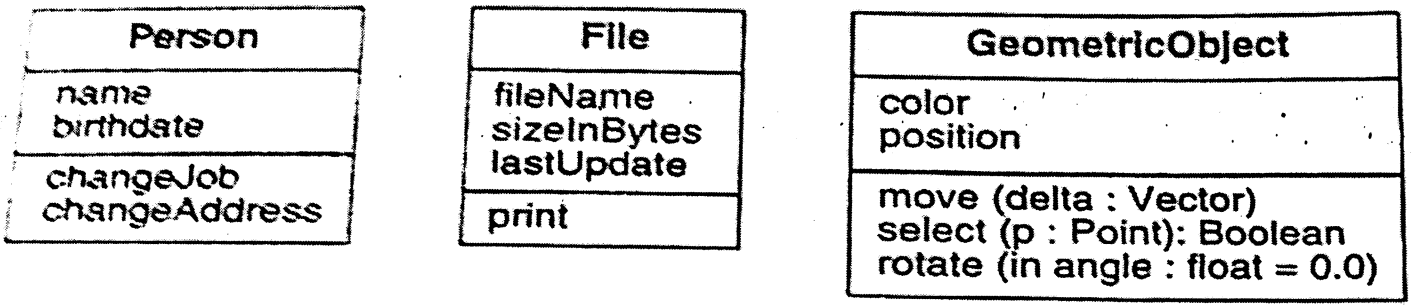


Figure 3.4 Operations. An operation is a function or procedure that may be applied to or by objects in a class.

Summary of Notation for Classes

- A box represents a class and may have as many as 3 compartments.
- The compartments contain, from top to bottom: class name, list of attributes and list of operations.
- Optional details such as type and default value may follow each attribute name.
- Optional details such as argument list & result type may follow each operation name.

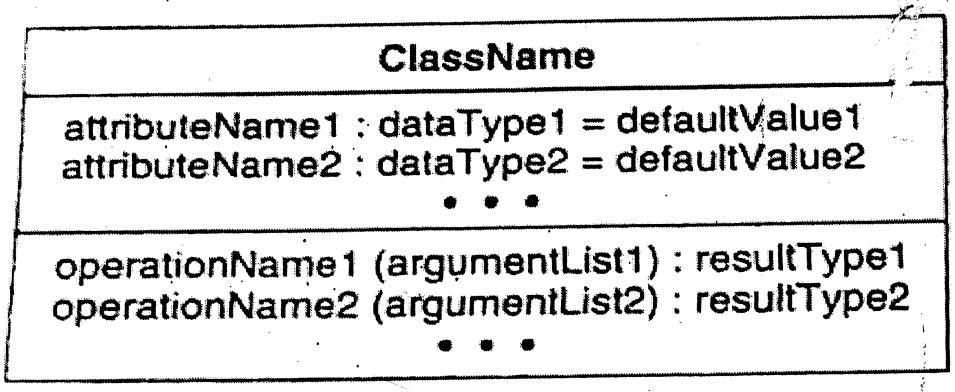


Figure 3.5 Summary of modeling notation for classes. A box represents a class and may have as many as three compartments.

- It shows 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.
- The attribute & operation compartments of class boxes are optional.

Link and Association Concepts

→ Links & associations are the means for establishing relationships among objects & classes.

Links and Associations

→ A link is a physical or conceptual connection among objects.
Ex. JoeSmith Works for Simplex company.

→ Most links relate two objects, but some links relate 3 or more 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.
Ex. a person WorksFor a company.

→ The links of an association connect objects from the same classes.

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

→ Links and associations often appear as verbs in problem statemen

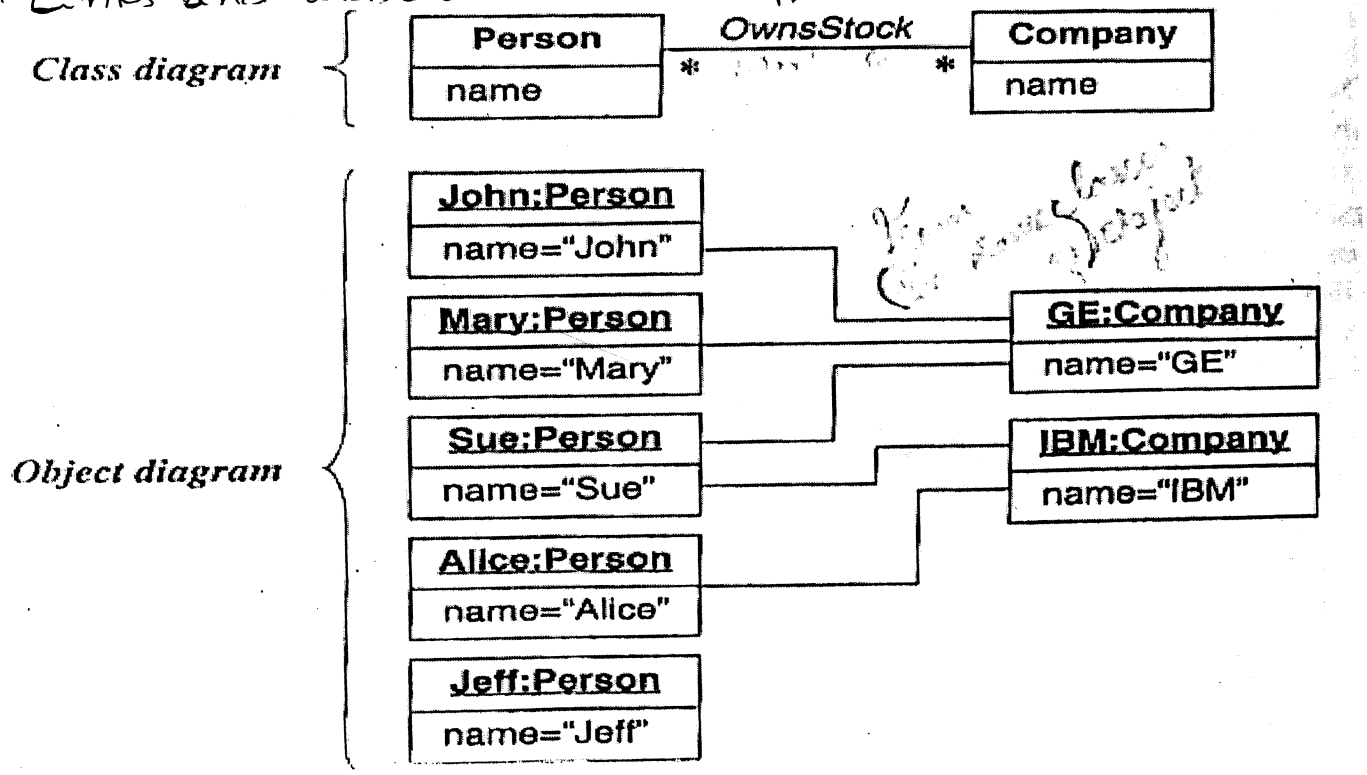


Figure 3.7 Many-to-many association. An association describes a set of potential links in the same way that a class describes a set of potential objects.

→ The top portion of the fig shows a class diagram & the bottom shows an object diagram.

→ In class diagram, a person may own stock in zero or more companies; a company may have multiple persons owning its stock.

→ In object diagram, John, Mary & Sue own stock in the GE. Sue & Alice own stock in the IBM company.

→ Jeff doesnot own stock in any company & thus has no link.

→ The asterisk (*) is a multiplicity symbol. Multiplicity specifies the no. of instances of one class that may relate to a single instance of another class.

→ The UML notation for a link is a line between objects. If the link has a name, it is underlined.

→ The association name is optional, if the model is unambiguous. Ambiguity arises when a model has multiple associations among the same classes (person works for company & person owns stock in company).

→ Associations are inherently bidirectional.

→ 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.

→ Associations are important, precisely because they break encapsulation. Associations cannot be private to a class, because they transcend classes.

→ Modeling treats associations as bidirectional, we do not have to implement them in both directions.

Multiplicity

→ Multiplicity specifies the no. of instances of one class that may relate to a single instance of an associated class.

→ Multiplicity constrains the no. of related objects. 1 or many

→ UML diagrams explicitly list multiplicity at the ends of association lines.

→ The UML specifies multiplicity with an interval, such as "1" (exactly one), "1...*" (one or more) or "3...5" (Three to five). The special symbol "*" is a standard notation that denotes "many" (zero or more).

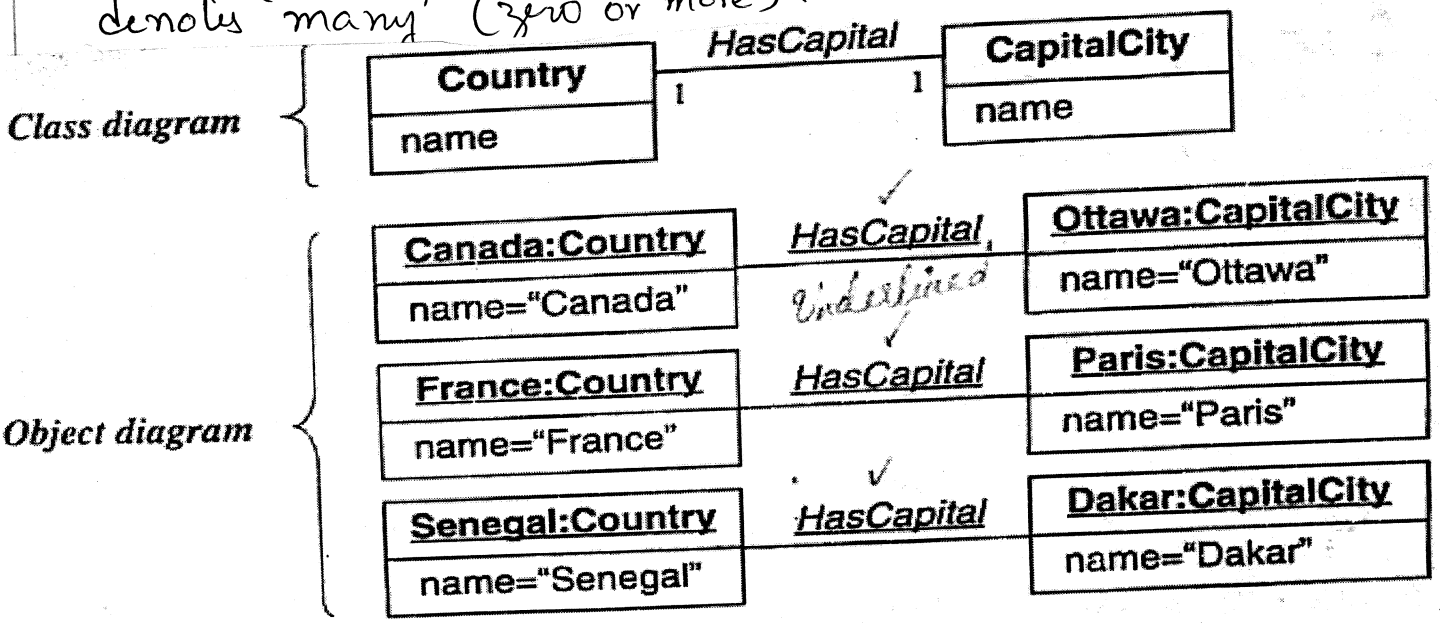


Figure 3.8 One-to-one association. Multiplicity specifies the number of instances of one class that may relate to a single instance of an associated class.

→ Fig shows one to one association and some corresponding links. Each country has one capital city. A capital city administers one country.

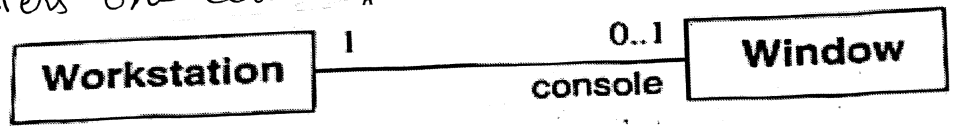


Figure 3.9 Zero-or-one multiplicity. It may be optional whether an object is involved in an association.

→ A workstation may have one of its windows designed as the console to receive general error messages. It is possible however, that no console window exists.

→ Multiplicity is a constraint on the size of a collection.
→ Cardinality is the count of elements that are actually in a collection.
→ ∴ Multiplicity is a constraint on the cardinality.
→ A multiplicity of "many" specifies that an object may be associated with multiple objects. For each association there is at most one link blw a given pair of objects
→ Fig shows if we want two links between the same objects, we must have two associations.

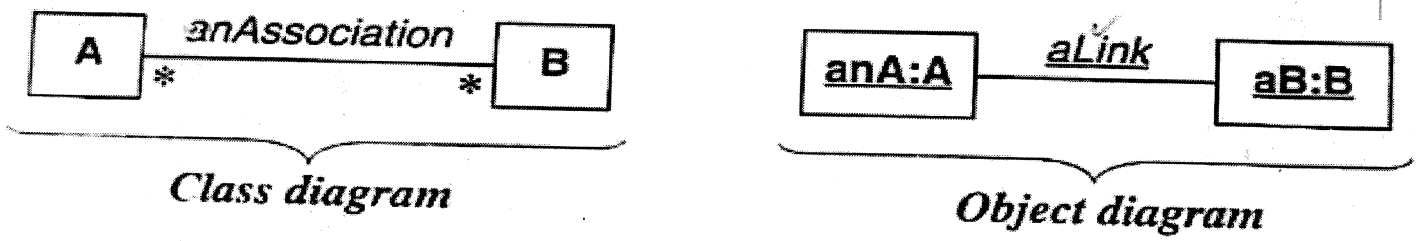


Figure 3.10 Association vs. link. A pair of objects can be instantiated at most once per association (except for bags and sequences).

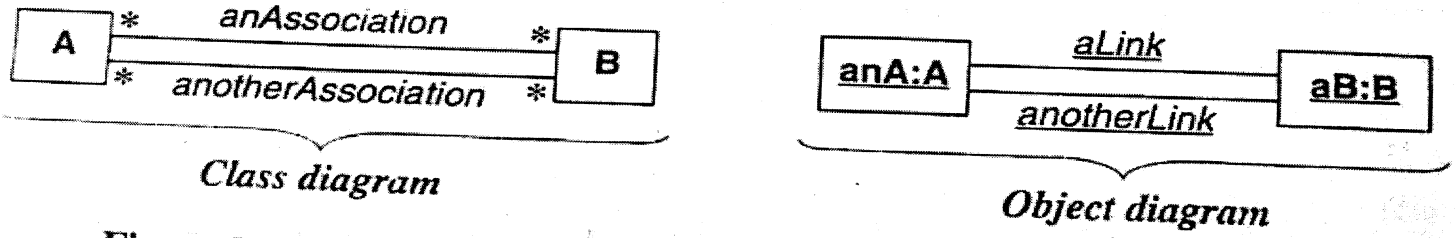


Figure 3.11 Association vs. link. You can use multiple associations to model multiple links between the same objects.

Association End Names

→ Multiplicity implicitly referred to the ends of associations
Eg:. 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.

→ Association end names often appear as nouns in problem descriptions.

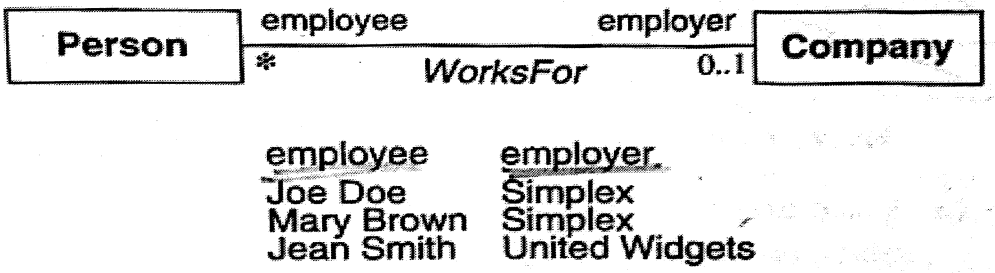


Figure 3.12 Association end names. Each end of an association can have a name.

→ As fig shows, a name appears next to the association end. Person & Company participate in association WorksFor.
→ A person is an employee with respect to a company; A company is an employer with respect to a person.
→ Use of association end names is optional, but it is often easier and less confusing to assign association end names instead of, or in addition to, association names.
→ Association end names are especially convenient for traversing associations, because we can treat each one as a pseudo attribute.
→ Association end names provide a means of traversing an association without explicitly mentioning the association.
→ Association end names are necessary for associations between two objects of the same class.
Ex: Fig Container & contents distinguish the two usages of Directory in the self association.
→ A directory may contain many lesser directories and may optionally be contained itself.
→ Association end names can also distinguish multiple associations b/w same pair of classes.

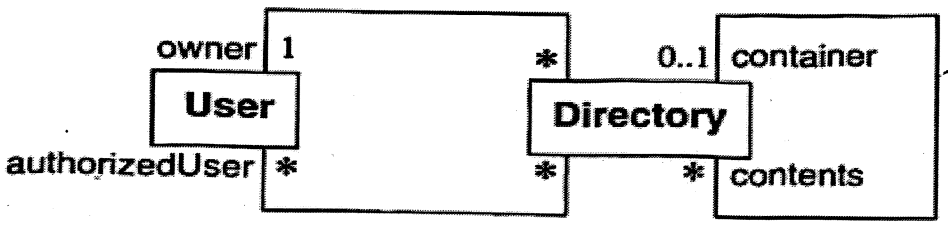


Figure 3.13 Association end names. Association end names are necessary for associations between two objects of the same class. They can also distinguish multiple associations between a pair of classes.

→ 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 b/w a pair of distinct classes, the names of the classes often suffice, & we may omit association end names.
→ Association end names let us unify multiple references to the same class.
→ When constructing class diagrams we should properly use association end names & not introduce a separate class for each reference.
→ Fig. In the wrong model, two instances represent a person with a child, one for the child & one for the parent.
→ In correct model, one person instance participates in 2 or more links, twice as a parent & zero or more times as a child.

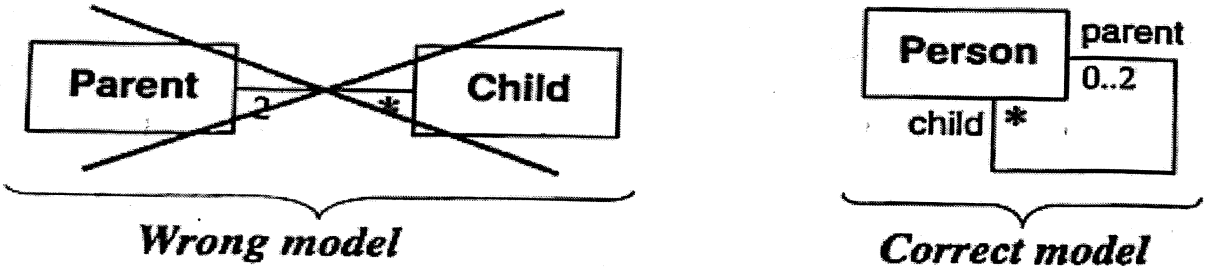


Figure 3.14 Association end names. Use association end names to model multiple references to the same class.

→ Association end names distinguish objects, all names on the far end of associations attached to a class must be unique.
→ No association end name should be same as an attribute name of the source class.

Ordering

→ The objects on a "many" association end have no explicit order, & we can regard them as a set. Sometimes the objects have explicit order.

Eg: fig shows a workstation screen containing a no. of overlapping windows. Each window on a screen occurs at most once. The windows have an explicit order, so only the topmost window is visible at any point on the screen.

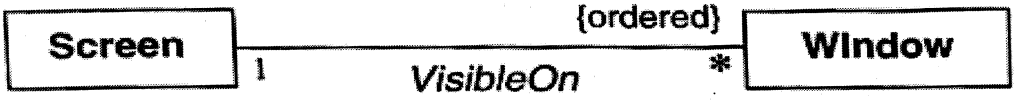


Figure 3.15 Ordering the objects for an association end. Ordering sometimes occurs for "many" multiplicity.

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

Bags and Sequences

→ A binary association has at most one link for a pair of objects.
→ 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.

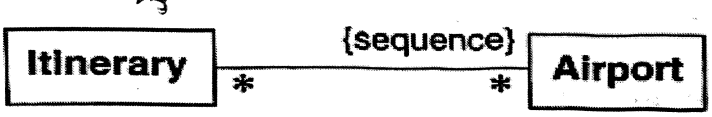


Figure 3.16 An example of a sequence. An itinerary may visit multiple airports, so you should use *{sequence}* and not *{ordered}*.

Eg. An itinerary is a sequence of airports and the same airport can be visited more than once.

↳ Like the {ordered} indication, {bag} and {sequence} are permitted only for binary associations.

→ If we specify {bag} or {sequence}, then there can be multiple links for a pair of objects. If we omit these annotations, then the association has at most one link for a pair of objects.

→ The {ordered} and the {sequence} annotations are the same, except that the first disallows duplicates and the other allows the

→ A sequence association is an ordered bag, while an ordered association is an ordered set.

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.

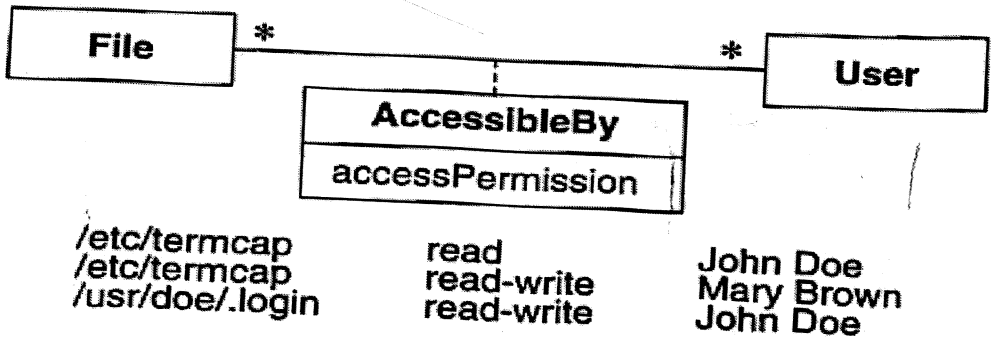


Figure 3.17 An association class. The links of an association can have attributes.

→ Fig.: accessPermission is an attribute of AccessibleBy.

→ The sample data at the bottom of the figure shows the value for each link.

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

→ In fig accessPermission is a joint property of File & User & cannot be attached to either File or User alone without losing information.

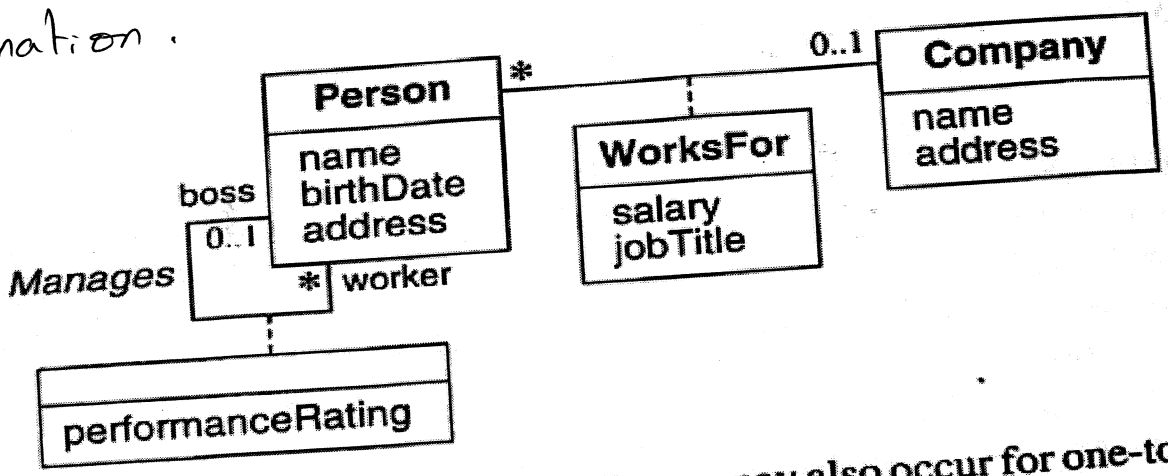
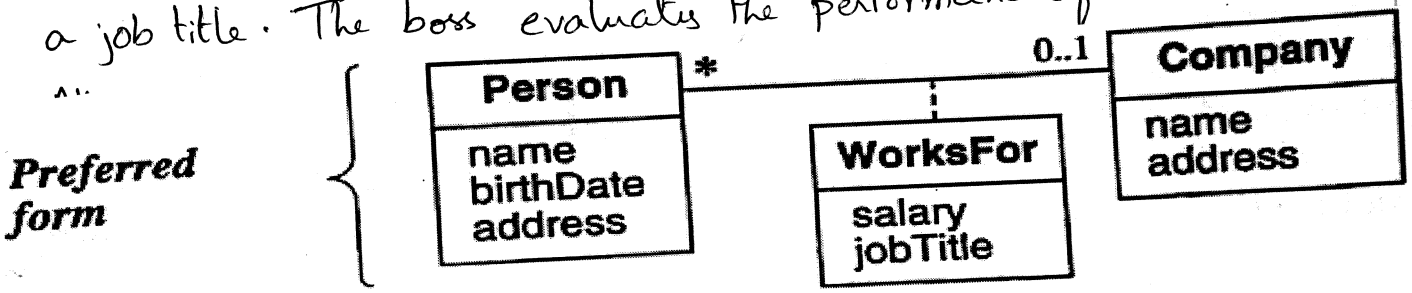


Figure 3.18 Association classes. Attributes may also occur for one-to-many and one-to-one associations.

→ In fig. present attributes for two one to many associations.

→ Each person working for a company receives a salary & has a job title. The boss evaluates the performance of each worker.



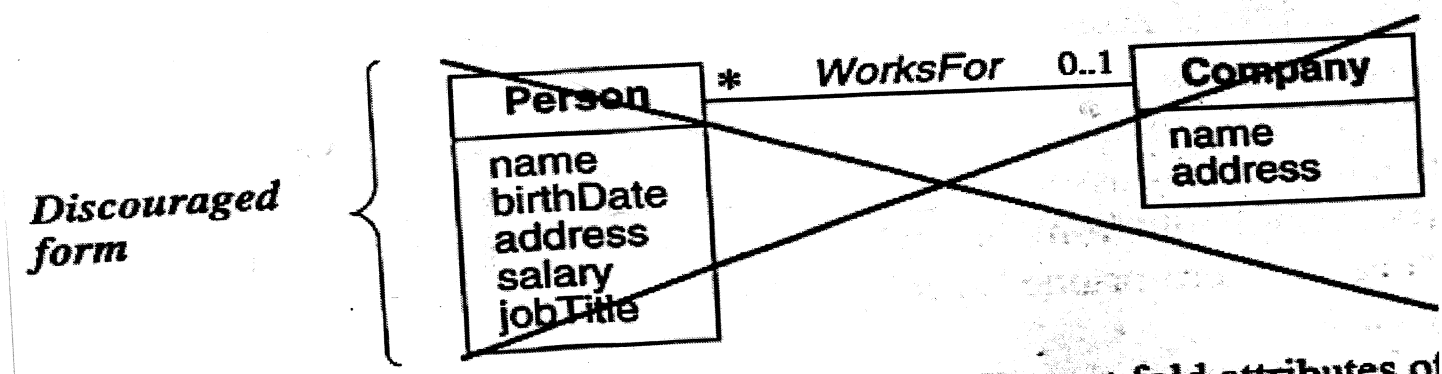


Figure 3.19 Proper use of association classes. Do not fold attributes of an association into a class.

→ fig shows how it is possible to fold attributes for one-to-one and one-to-many associations into the class opposite a one e This is not possible for many to many associations.

→ As a rule, we should not fold such attributes into a class because the multiplicity of the association might change.

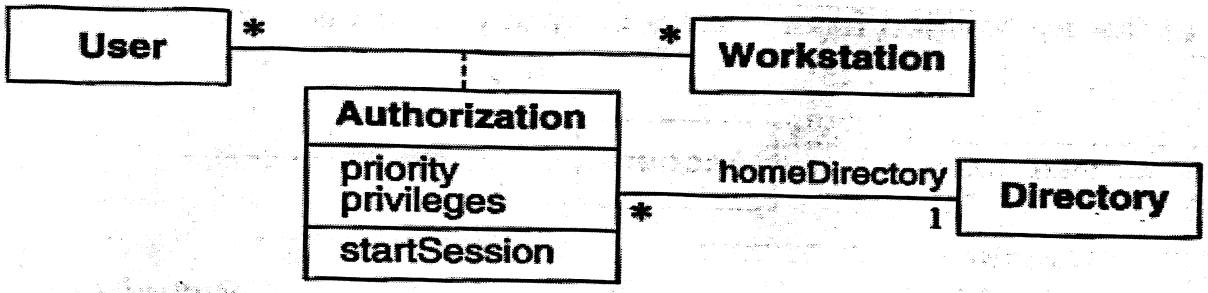


Figure 3.20 An association class participating in an association. Association classes let you specify identity and navigation paths precisely.

→ fig 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 director

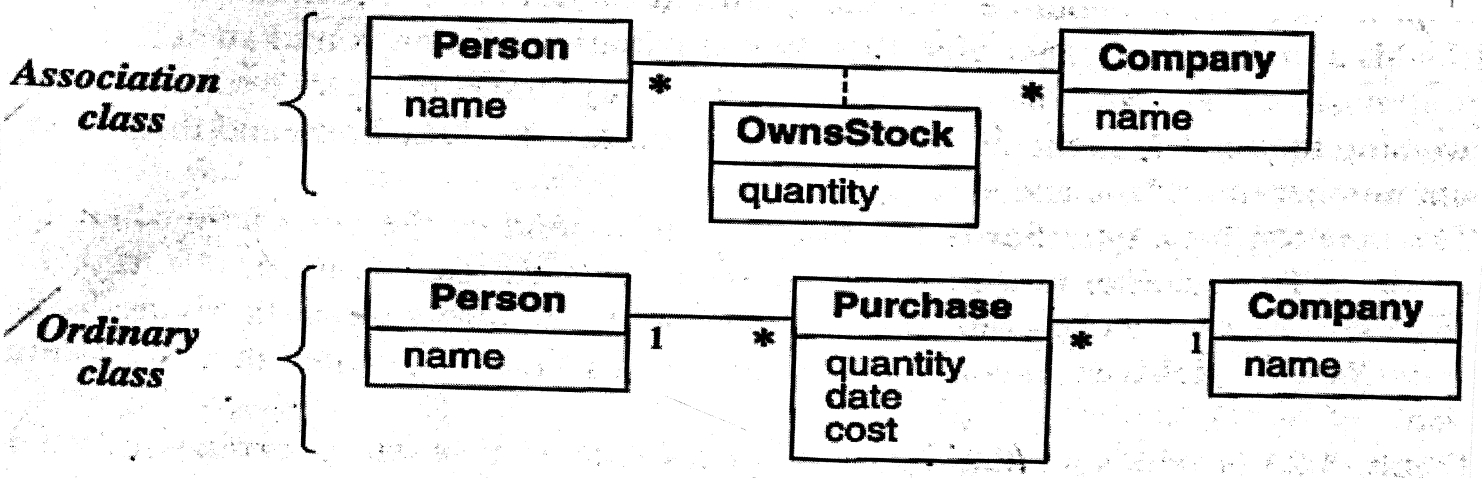


Figure 3.21 Association class vs. ordinary class. An association class is much different than an ordinary class.

→ The association class has only one occurrence for each pairing of Person & Company. In contrast there can be any no. of occurrences of a Purchase for each Person & Company.
→ Each purchase is distinct and has its own quantity, date, & cost.

Qualified Associations -

→ A qualified association is an association in which an attribute called the qualifier disambiguates the objects for a "many" association end. → it selects objects from a larger set of related objects

→ It is possible to define qualifiers for one-to-many & many-to-many associations.

→ A qualifier selects among the target objects, reducing the effective multiplicity, from "many to one".

→ Qualified associations with a target multiplicity of "one" or "zero or one" specify a precise path for finding the target object from the source object.

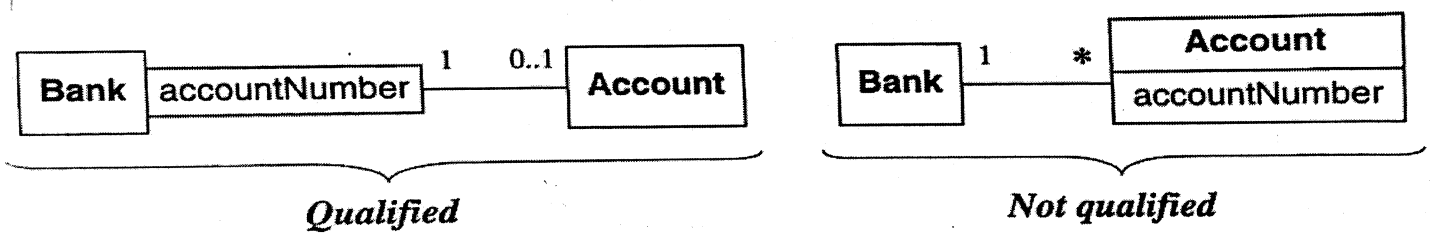


Figure 3.22 Qualified association. Qualification increases the precision of a model.

→ A bank services multiple accounts. An account belongs to a single bank. Within the context of a bank, the account no. specifies a unique account.

→ Bank and Account are classes and accountNumber is the qualifier.

→ Qualification reduces the effective multiplicity of this association from one-to-many to one-to-one.

→ Both models are acceptable, but the qualified model adds information. The qualified model adds a multiplicity constraint, that the combination of a bank & an account no. yields at most one account.

→ The notation for a qualifier is a small box on the end of the association line near the source class.
→ The qualifier box may grow out of any side (top, bottom, left, right) of the source class.
→ The source class plus the qualifier yields the target class.
→ In fig. Bank + accountNumber yields an Account, therefore accountNumber is listed in a box contiguous to Bank.

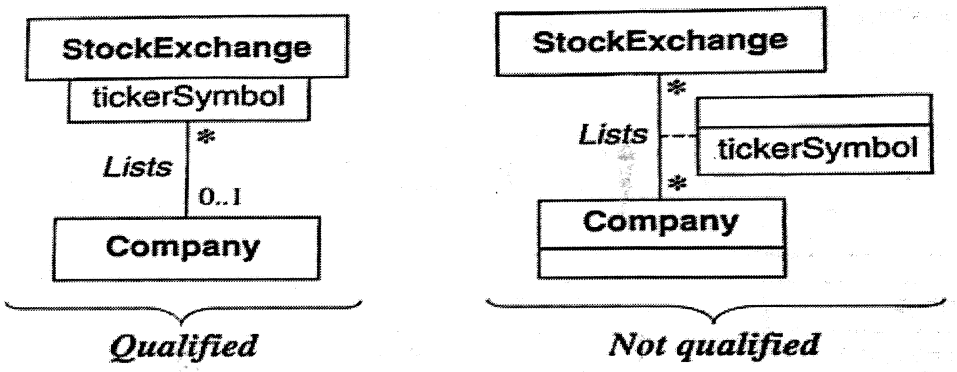


Figure 3.23 Qualified association. Qualification also facilitates traversal of class models.

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

Generalization and Inheritance

Definition

→ Generalization is a relationship b/w a class (the superclass) and one or more variations of the class (the subclass).
→ Generalization organizes classes by their similarities & differences, structuring the description of objects.

→ The superclass holds common attributes, operations, & 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.

→ Generalization organizes classes into a hierarchy; each subclass has a single immediate superclass. There can be multiple levels of generalization.

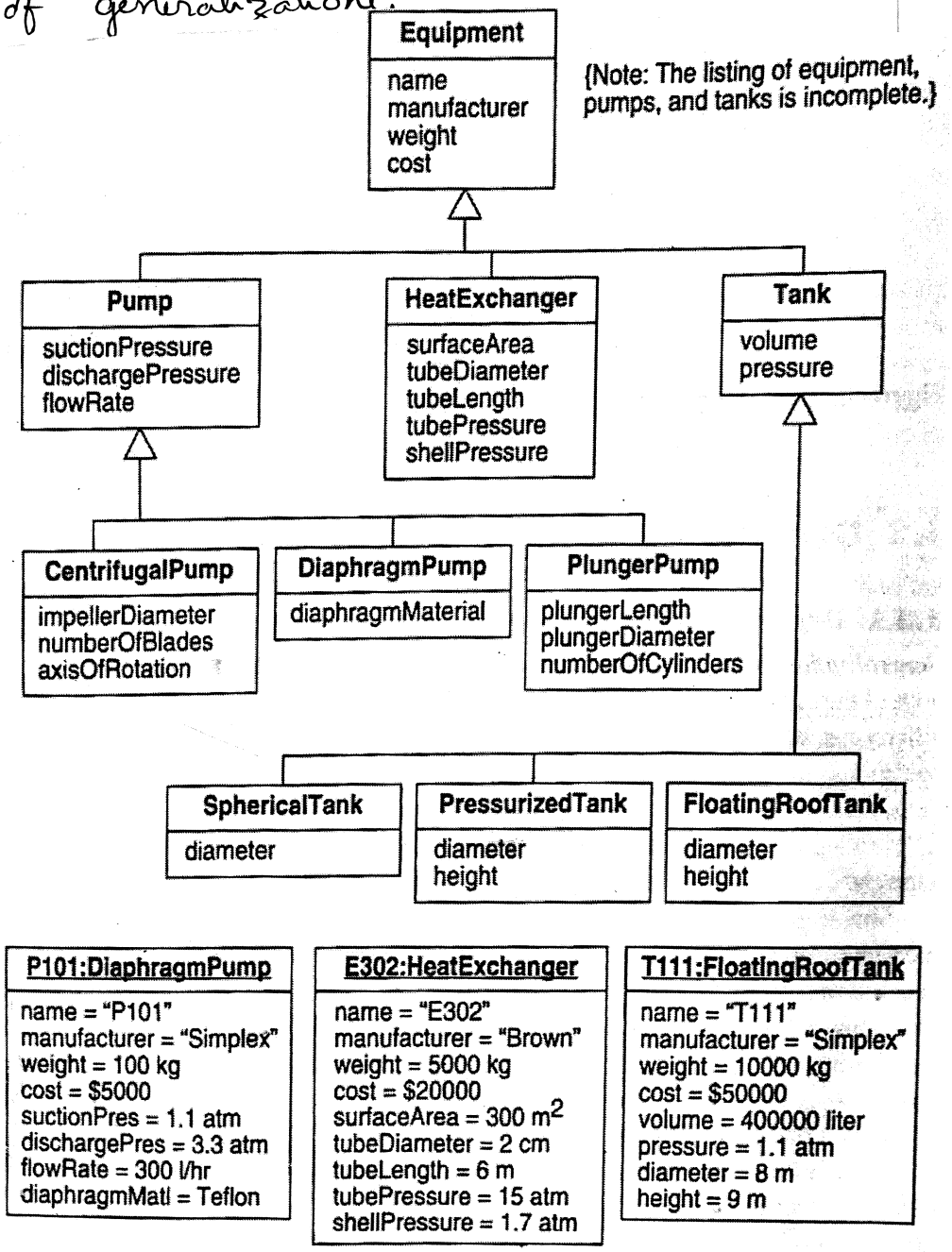


Figure 3.24 A multilevel inheritance hierarchy with instances. Generalization organizes classes by their similarities and differences, structuring the description of objects

Ex: Generalization for equipment. Each piece of equipment is a pump, heat exchanger, or tank.

→ There are several kinds of pumps: centrifugal, diaphram, & plunger.

→ There are several kinds of tanks: spherical, pressurized, & floating roof.

→ The fact that the tank generalization symbol is drawn below the pump generalization symbol is not significant.

→ Several objects are displayed at the bottom of the figure. Each object inherits features from one class at each level of the generalization.

→ P101 embodies the features of equipment, pump, & diaphragm pu
E302 has the properties of equipment & heat exchanger.

→ A large hollow arrowhead denotes generalization. The arrowhead points to the super class. Generally draw the superclass on top & the subclasses on the bottom.

→ The curly braces denote a UML comment, indicating that there are additional subclasses that the diagram does not show.

→ Generalization is transitive across an arbitrary no. 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.

→ An instance includes a value for every attribute of every ancestor class. An instance can invoke any operation on any
→ ancestor class.

→ Each subclass not only inherits all the features of its ancestors but adds its own specific features as well.

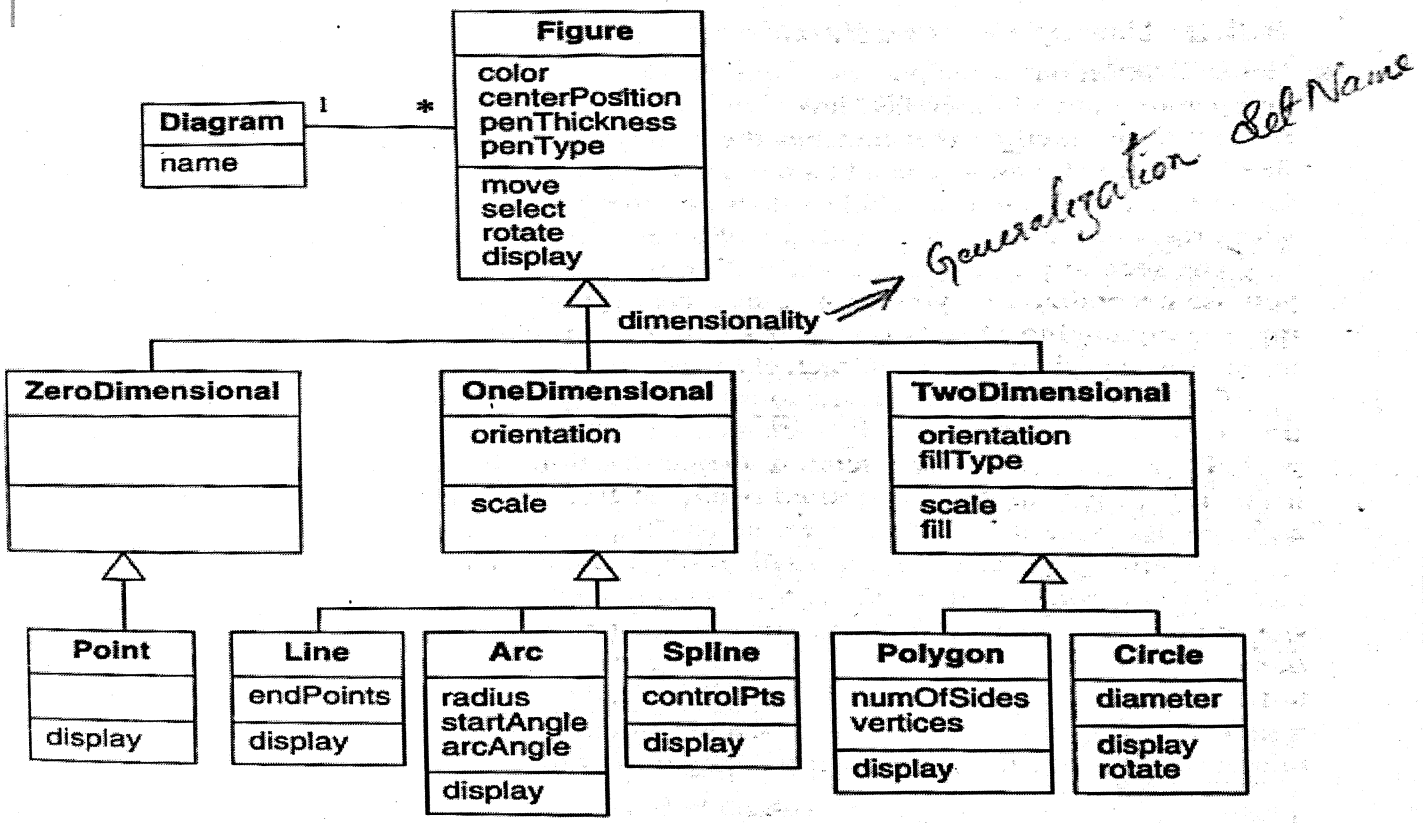


Figure 3.25 Inheritance for graphic figures. Each subclass inherits the attributes, operations, and associations of its superclasses.

→ It shows classes of geometric figures. Move, select, rotate, & display are operations that all subclasses inherit. Scale applies to one-dimensional & two-dimensional figures. Fill applies only to two-dimensional figures.

→ 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.

→ Generalization set values are inherently in one-to-one correspondence with the subclasses of a generalization.

→ The generalization set name is optional.

→ An inheritance hierarchy that is two or three levels deep is certainly acceptable; ten levels deep is probably excessive; five or six levels may or may not be proper.

Use of Generalization

Generalization has 3 purposes.

(1) Support for polymorphism

→ We can call an operation at the superclass level, & the OO language compiler automatically resolves the call to the method that matches the calling objects' class.

Polymorphism increases the flexibility of s/w: we can add a new subclass & automatically inherit superclass behavior.

(2) Generalization is to structure the description of objects

When we are using generalization, - we are forming a taxonomy & organizing objects on the basis of their similarities & differences. This is much more greater than modeling each class individually & in isolation from other classes.

(3) Generalization to enable reuse of code

we can inherit code within our application as well as from past work such as a class library. Reuse is more productive than repeatedly writing code from scratch. Generalization also lets us adjust the code, where necessary, to get the precise derived behavior.

Generalization & specialization concern a relationship among classes and take opposite perspectives, viewed from the superclass or from the subclasses.

→ The word generalization derives from the fact that the superclass generalizes the subclasses.
→ Specialization refers to the fact that the subclasses refine or specialize the superclass.
→ Inheritance is the mechanism for sharing attributes, operations & associations via the generalization/specialization relationship.

Overriding features

→ A subclass may override a superclass feature by defining a feature with the same name.
→ The overriding feature (the subclass feature) refines & replaces the overridden feature (the superclass feature).
→ There are several reasons to override a feature:
  ↳ To specify behavior that depends on the subclass,
  ↳ To tighten the specification of a feature, or ~~to~~ imp.
  ↳ To improve performance.

Ex:- in fig. each leaf subclass must implement display, even though Figure defines it.

→ We may override methods and default values of attributes we should never override the signature, or form, of a feature.
→ An override should preserve attribute type, no. & type of arguments to an operation, & operation return type.
→ We should never override a feature so that it is inconsistent with the original inherited feature.
→ A subclass is a special case of its superclass and should be compatible with it in every respect.

A Sample Class Model

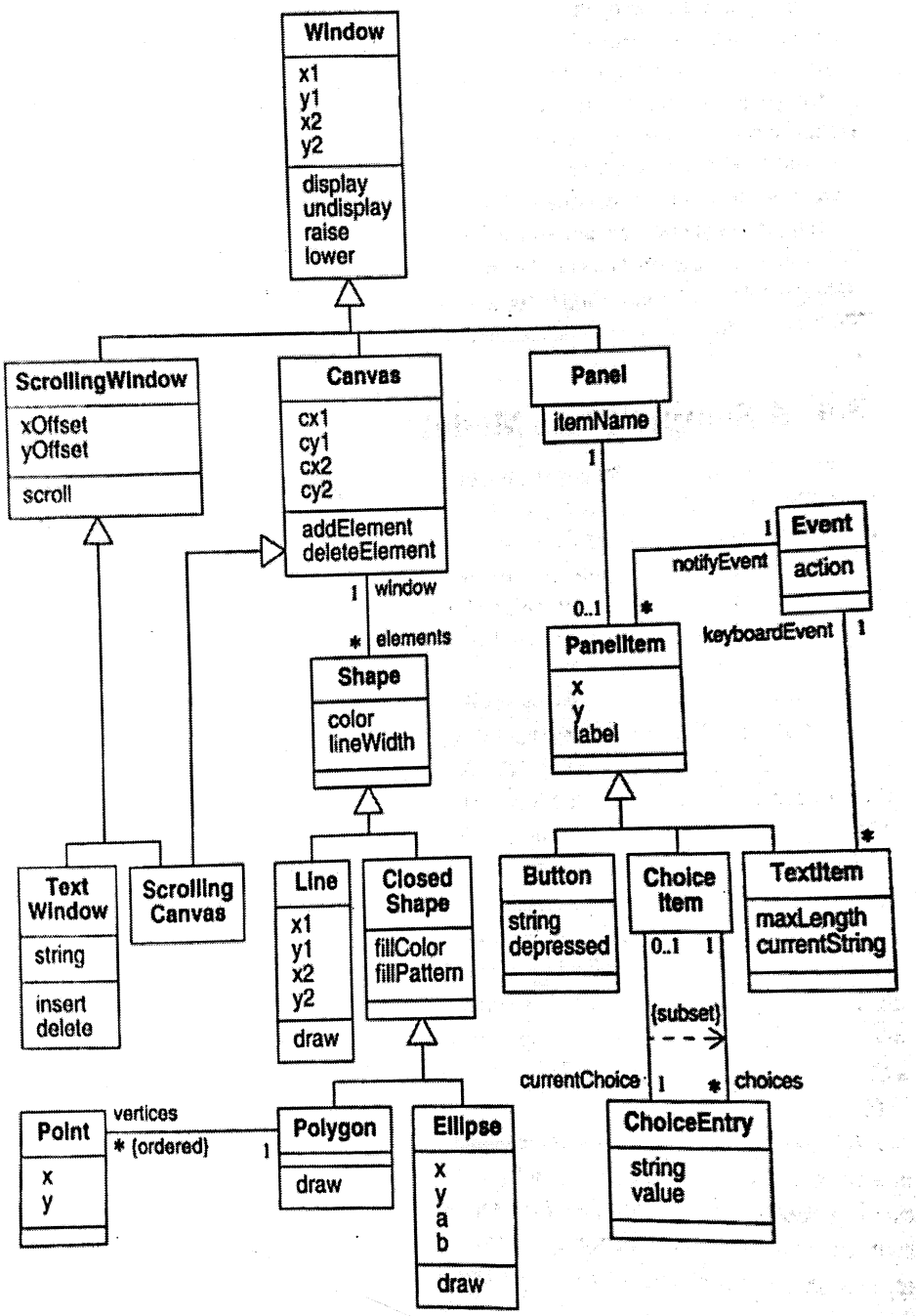


Figure 3.26 Class model of a windowing system

- Fig. shows a class model of a workstation window management system.

→ Class Window defines common parameters of all kinds of windows, including a rectangular boundary defined by the attributes x_1, y_1, x_2, y_2 and operations to display & undisplay a window & to raise it to the top or lower it to bottom of entire set of windows.

→ A canvas is a region for drawing graphics. It inherits the window boundary from Window and adds the dimensions of the underlying canvas region defined by attributes cx_1, cy_1, cx_2, cy_2.

→ A canvas contains a set of elements, shown by the association to class Shape.

→ All shapes have color and line width.

→ Shapes can be lines, ellipses, or polygons, each with their own parameters.

→ A polygon consists of a list of vertices. Ellipses and polygons are both closed shapes, which have a fill color and a fill pattern. Lines are one dimensional and cannot be filled.

→ Canvas windows have operations to add and delete elements.

→ TextWindow is a kind of a ScrollingWindow, which has a two dimensional scrolling offset within its window, as specified by xOffset and yOffset, as well as an operation scroll to change the scroll value.

→ A text window contains a string and has operations to insert and delete characters.

→ ScrollingCanvas is a special kind of canvas that supports scolling: it is both a canvas and a ScrollingWindow. This is an example of multiple inheritance.

→ A Panel contains a set of PanelItem objects, each identified by a unique itemName within a given panel, as shown by the qualified association.

→ Each panel item belongs to a single panel. A panel item is a predefined icon with which a user can interact on the screen.

→ Panel items come in 3 kinds: buttons, choice items, & text items.
→ A button has a string that appears on the screen; a button can be pushed by the user and has an attribute depressed.
→ A choice item allows the user to select one of a set of predefined choices, each of which is a ChoiceEntry containing a string to be displayed and a value to be returned if the entry is selected.
→ There are 2 associations b/w ChoiceItem and ChoiceEntry a one-to-many association defines the set of allowable choices, while a one-to-one association identifies the current choice.
→ The current choice must be one of the allowable choices, so one association is a subset of the other as shown by the arrow b/w them labeled "{subset}".
→ When a panel item is selected by the user, it generates an Event, which is a signal that something has happened together with an action to be performed.
→ All kinds of panel items have notifyEvent associations.
→ Each panel item has a single event, but one event can be shared among many panel items.
→ Text items have a second kind of event, which is generated when a keyboard character is typed while the text item is selected. The association with end name keyboardEvent shows these events.
→ The Text items also inherit the notifyEvent from superclass PanelItem; the notifyEvent is generated when the entire text item is selected with a mouse.

Navigation (flow) of Class Models

→ Navigation is important because it lets us exercise a model and uncover hidden flaws and omissions so that we can repair them.

→ We can perform navigation manually (informal technique) or write navigation expressions

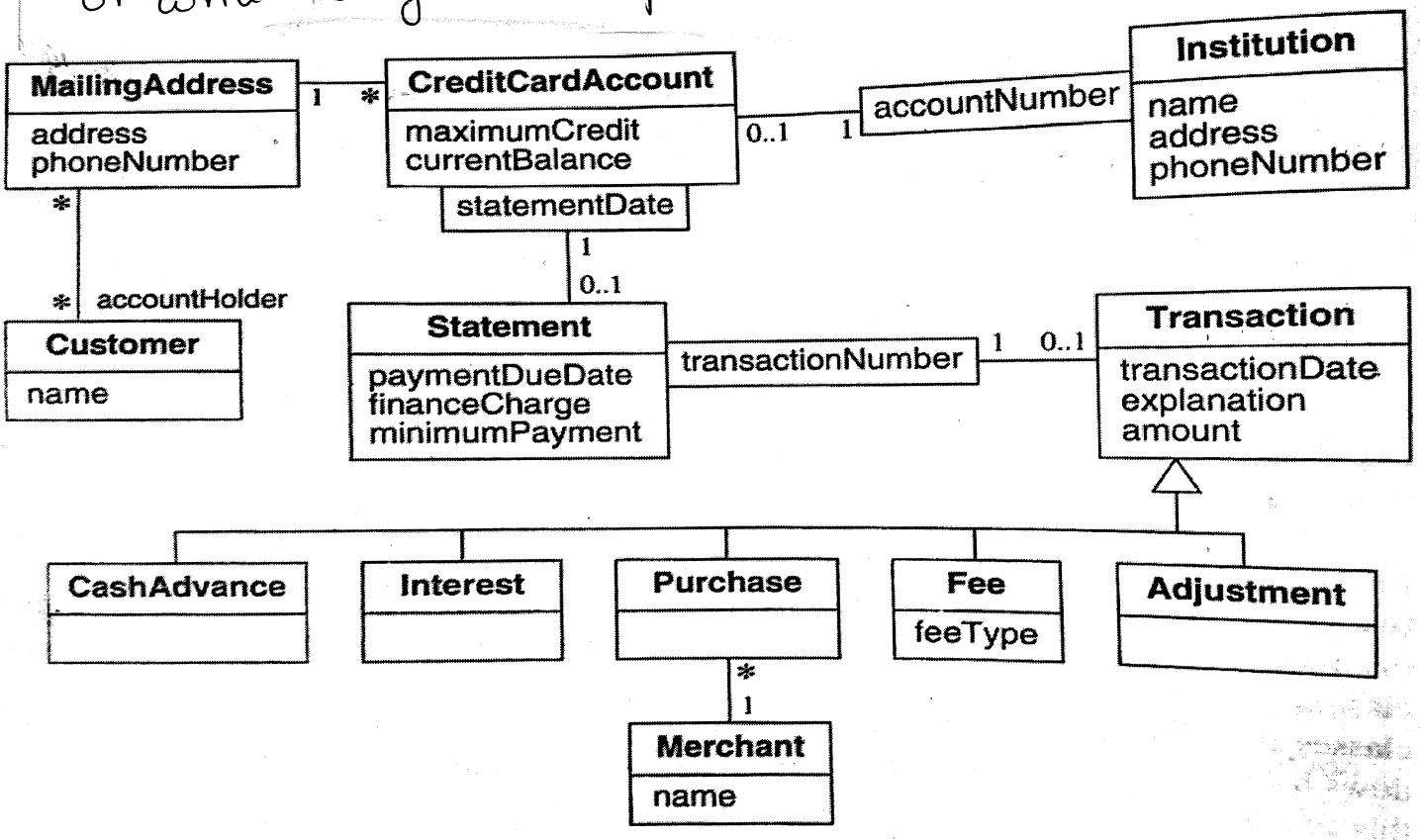


Figure 3.27 Class model for managing credit card accounts

→ An institution may issue many credit card accounts, each identified by an account no.. Each account has a maximum credit limit, a current balance, & a mailing address. The account serves one or more customers who reside at the mailing address.

→ The institution periodically issues a statement for each account. The stmt. lists a payment due date, finance charge, & min. payment.

→ The stmt. itemizes various transactions that have occurred throughout the billing interval: cash advances, interest charges, purchases, fees & adjustments to the account. The name of merchant is printed for each purchase.

We can pose a variety of questions against the model.
* What transactions occured 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?

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

OCL Constructs for Traversing Class Models

The OCL can traverse the constructs in class models.

* Attributes: We can traverse from an object to an attribute value. The syntax is the source object, followed by a dot, & then the attribute name.
Ex: aCreditCardAccount.maximumCredit

* Operations: We can also invoke an operation for an object or a collection of objects. The syntax is the source object or object collection, followed by a dot, & then the operation. An operation must be followed by parantheses, even if it has no arguments, to avoid confusion with attributes.

→ The OCL has special operations that operate on entire collections. The syntax for a collection operation is the source object collection, followed by "→", and then the operation.

* Simple associations: The dot notation is used to traverse an association to a target end.
The target end may be indicated by an association end name or, where there is no ambiguity, a class name.
Ex., aCustomer.MailingAddress yields a set of addresses for a customer (target end has "many" multiplicity).
aCreditCardAccount.MailingAddress yields a single address (target end has multiplicity of one).

* Qualified associations: A qualifier makes a more precise traversal.
→ The expression aCreditCardAccount.Statement[30November1999] finds the statement for a credit card account with stmt. date of 30 Nov 1999. The syntax is to enclose the qualifier value in brackets.
→ The expression aCreditCardAccount.Statement finds the multiple statements for a credit card account. (The multiplicity is "many" when the qualifier is not used)

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

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

* Filters: There is often a need to filter the objects in a set. The OCL has several kinds of filters, the most common of which is the select operation.
Ex. aStatement.Transaction→select(amount>$100)
It finds the transactions for a statement in excess of $100.

Building OCL Expressions

→ The real power of the OCL comes from combining primitive constructs into expressions.

→ An OCL expression could chain together several association traversals. There could be several qualifiers, filters, & operators aswell.

→ With the OCL, a traversal from an object through a single association yields a singleton or a set. A traversal through multiple associations can yield a bag.

→ A set is a collection of elements without duplicates.

→ A bag is a collection of elements with duplicates allowed.

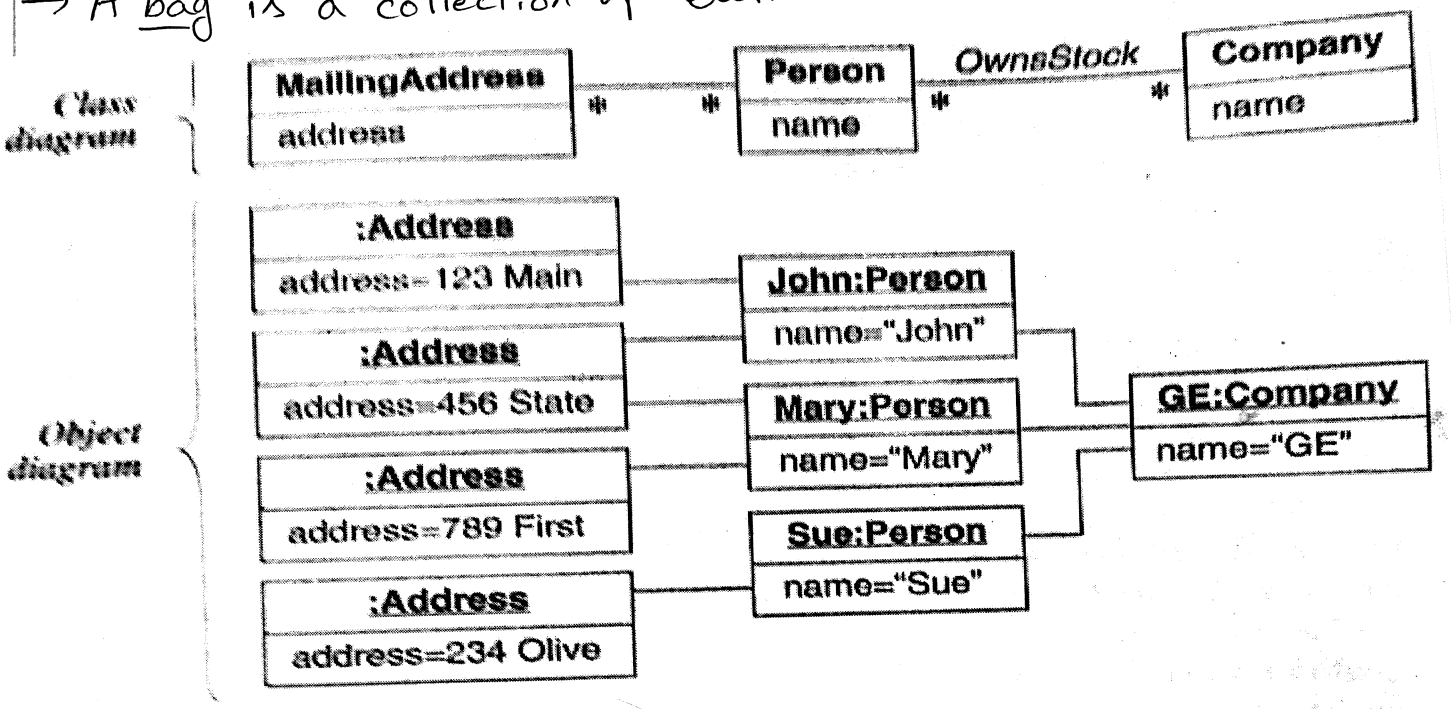


Figure 3.28 A sample model and examples. Traversal of multiple associations can yield a bag.

→ Fig illustrates how an OCL expression can yield a bag. A company might want to send a single mailing to each stockholder address. Starting with the GE company, we traverse the OwnsStock association and get a set of three persons. Starting with these 3 persons & traversing to mailing address, we get a bag obtaining the address 456 State twice.

Examples of OCL Expressions

We can use the OCL to answer the credit card quations.

* What transactions occured for a credit card account within a time interval?

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

→ The expression traverses from a CreditCardAccount object to Statement and then to Transaction, resulting in a set of transactions.

→ The OCL select operator used to find the transactions within the time interval bounded by aStartDate and anEndDate.

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

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

→ The expression traverses from an Institution object to CreditCardAccount, then to Statement, and then to Transaction.

→ The OCL select operator finds the transactions within the time interval bounded by aStartDate and anEndDate.

→ Then find the amount for each transaction and compute the total with the OCL sum operator.

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

```
aMerchant.Purchase -> select(aStartDate <= transactionDate and
transactionDate <= anEndDate).Statement.CreditCardAccount.
MailingAddress.Customer -> asSet()
```

→ The expression traverse from a Merchant object to Purchase

→ The OCL select operator finds the transactions within the time interval bounded by a aStartDate and anEndDate.

→ Then traverse to Statement, then to CreditCardAccount, then to MailingAddress, and finally to Customer.

→ The association from MailingAddress to Customer is many to many (bag)

→ The OCL asSet operator converts a bag of customers to a set of customers.

* How many credit card accounts does a customer currently have?

aCustomer.MailingAddress.CreditCardAccount→size()

→ Given a Customer object, we find a set of MailingAddress objects. Then given a set of MailingAddress objects, we find a set of CreditCardAccount objects.

→ For the set of CreditCardAccount objects we apply the OCL size operator, which returns the cardinality of the set.

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

aCustomer.MailingAddress.CreditCardAccount.maximumCredit→sum()

→ The expression traverse from a Customer object to MailingAddress and then to CreditCardAccount, yielding a set of CreditCardAccount objects.

→ For each CreditCardAccount, we find the value of maximumCredit and compute the total with the OCL sum operator.

* These kinds of questions exercise a model and uncover hidden flaws and omissions that can then be repaired.

* The OCL was originally intended as a constraint language.