MODULE : 2 CH: 1 Introduction

OOMD: is a way of thinking about problems using models organized around real-world concepts.

(*) What is Object Orientation?

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

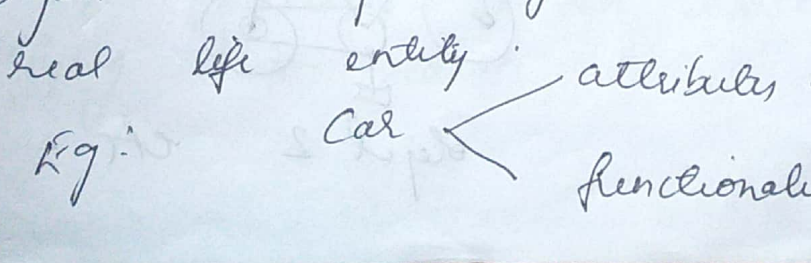
Characteristics of OO approach are:

1. Identity
2. Classification
3. Inheritance
4. Polymorphism

Identity: means that data is quantized into discrete, distinguishable entities called objects. Each object has its own inherent identity. Two objects are distinct even if all their attribute values are identical.

Object is a piece of code which represent the real life entity.

Eg:



Classification: means that object with the same datastructure (attributes) & behaviours (operations) are grouped into a class. **Class** is a collection of object. Each class describes infinite set of individual object.

Eg: Class Student

```
{
    RNo ;
    Name ;
    marks ;
    CA() ;
    RCA() ;
}
```

Specification of objects.

Each object is said to be an instance of a class.

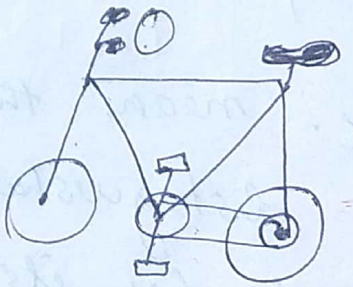
Eg: Object and Class - Each class describes infinite set of individual objects.

Bicycle class

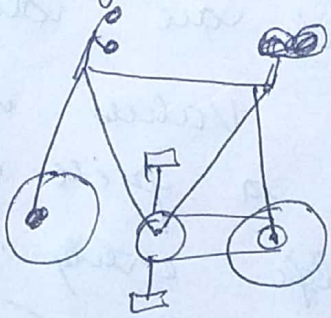
Attributes:
- Frame Size
- wheel size
- number of gears
- material

Operations
- Shift
- move
- repair

Bicycle Objects



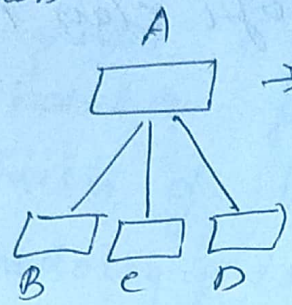
object 1



object 2 etc.

Inheritance: is the sharing of attributes & operations among classes based on a hierarchical relationship. i.e Acquiring the features of base class.

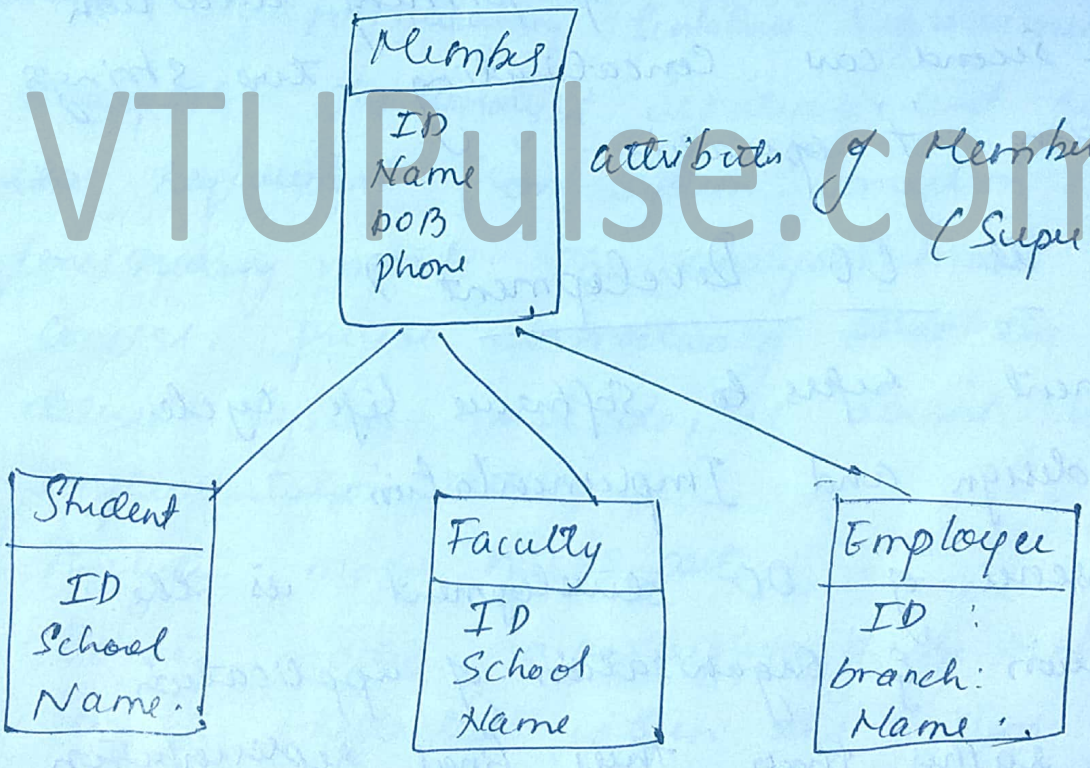
Eg:



→ Super class or base class.

→ Derived (or) or Sub class.

All derived class inherit the attributes of Super class, this process is inheritance.



attributes of Member class (Super class)

=> Base class are Student Faculty Employee.

This is mainly reusability: we can add additional features to an existing class without modifying that class.

Polymorphism: means that the same operation may behave differently for different classes. operation is a procedure, implementation of an operation by a specific class is called a method.

Eg: operation of addition

(i) 3 + 4 = 7

(ii) Rama + Krishna = Ramakrishna

In first case we are performing addition and in second case concatinating two strings using same '+' operation.

(*) What is OO Development?

→ Development refers to software life cycle analysis, design and Implimentation.

→ The essence of OO development is the identification & organisation of application concepts, rather than their final representation in a programming language.

→ OO development is a way of thinking & not a programming technique; & it is a conceptual process

→ It serve as a medium for Specification

analysis, documentation & interfacing, as well as for programming.

OO Methodology

→ We present a process for OO development & a graphical notation for representing OO concepts. The process consists of building a model.

The methodology is as following stages.

- System Conception: Software development begins with business analysts or users conceiving an application & formulating tentative requirements.
- Analysis: The analyst scrutinizes and restates the requirement from system conception by constructing models. The analysis model is a concise, precise abstraction of what the desired system must do, it doesn't contain implementation decisions.

  Analysis model has 2-parts:

  Domain model - a description of the real-world objects reflected within the System.

  Application model - description of parts of the application System itself that are visible to the user.
- System design - The development team plan a high level strategy - the System architecture for solving the application problem.

They also establish policies that will serve as a default for subsequent, more detailed portions of designs.

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

- Class design: adds details to analysis model. 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 the system remains flexible and extensible.

Three Models

We use 3 kinds of models to describe system from different viewpoints

- Class model
- State model
- Interaction model.

Class model describes the static structure of the objects in a system and their relationships. Class model defines the content for s/w development. This contain class diagram: is a graph whose nodes are classes & arcs are relationship among classes.

State model describes the aspects of an object that change overtime. The state model specifies and implements control with state diagrams. State diagram is a graph whose nodes are state and whose arcs are transition b/w states caused by events.

Interaction mode describes how the objects in a system cooperate to achieve broader results. This starts with use cases: this focuses on the functionality of a system. Sequence diagram shows the objects that interact & the time sequence of their interactions. The activity diagram elaborates important processing.

(*) OO Themes:

Abstraction: focus on essential aspects of an application while ignoring details. It about what an object is & does, before deciding how to implement it.

Encapsulation: Separates the external aspects of an object, that are accessible to other object. from the internal implementation details, that are hidden from other objects. Encapsulation prevent portions of program from becoming so interdependent that a small change has massive ripple effects. Encapsulation is wrapping up of methods and data into a single unit.
It is also known as information hiding concept.

Combining Data & Behaviours: The caller of an operation need not consider how many implementations exist. The OO program simply invoke the draw operation on each figure (polygon, circle or text) 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.

Sharing: Inheritance of both data structure & behaviours lets subclasses share common code.
It (oo development) also provides reusing designs & code on future project.

Emphasis & Essence of an Object:

OO technology stresses what an object is, rather than how it is used. The uses of an object

depend on the details of the application & often change during development.

Synergy: The characteristics of OO language are Identity, classification, polymorphism & inheritance. Each of these concepts can be used in isolation.

The emphasis on the essential properties of an object forces the developer to think more carefully & deeply about what an object is & does.

Evidence for Usefulness of OO Development

→ OO development work began with internal application at General Electric Research & development centre.

→ OO techniques were used to develop compilers, graphics, user interfaces, databases, an OO language, CAD Systems, Simulations etc.

OO Modeling History:

→ The work at GE R&D led to the development of the Object Modeling Technique (OMT), which led to a problem - a plethora of alternative notations.

→ Expressed Similar ideas of different Symbols which led to confusing the developers

→ In 1996 OMG (Object Management Group) issued a request for proposals for a standard OO modeling notation.

The OMG unanimously accepted the resulting UML (Unified Modeling Language) as a standard in Nov, 1997.

→ UML was highly successful & replaced other notations.

CH. 2 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: Recent advances in computation permit the simulation of many physical structures without the need to build physical models. Both physical & computer models are usually cheaper than building a complex system & enable early

correction of faults.

Communication with customers: Architects & product designers build models to show their customers. This imitates some or all of the external behaviour of a system.

Visualization: Storyboards of movies, television shows & advertisements let writers see how their idea flow. They modify awkward transitions, dangling ends and unnecessary segments before detailed writing begins.

Reduction of Complexity: The reason of modeling is deal with systems that are too complex to understand directly.

* 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 & suppress those aspects that are unimportant.

All abstractions are incomplete & inaccurate.

Abstraction is all of representing essential features without including the background details

* The Three Models

The three kinds of models separate a system into distinct views. Different models have limited and explicit interconnections.

Class Model: describes the structure of objects in a system - their identity, their relationship to other objects, their attributes and their operations. The class models provides content for the state and interaction models.

The goal here is to construct a class model to capture those concepts from the real world that are important to an application. Class diagrams express the class model. Classes share structure & behaviour & associations relate the classes. classes define the attribute values carried by each object & the operations that each object performs or undergoes.

State Model:

State model describes those aspects of objects concerned with time & the sequencing of operations - events that mark changes, states that define the context for events and the organization of events & states. The state model captures control, the aspect

of a system that describes the sequence of operations that occur, without regard for what the operations do, what they operate on or how they are implemented.

State diagrams express the state model. Each state diagram shows the state & event sequences permitted in a system for one class of objects. Actions and events in a state diagram become operations on objects in the class model. References b/w state diagrams become interactions in the interaction model.

Interaction Model: This describes interactions b/w objects - how individual objects collaborate to achieve the behaviour of the system as a whole. The state & interaction model describe different aspects of behavior.

Use cases, sequence diagrams, & activity diagrams document the interaction model. Use cases document major themes for interaction b/w the system & outside actors. Sequence diagram show the objects that interact & the time sequence of their interactions. Activity diagram 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 the data structure on which the state & interaction models operate. The operations in the class model correspond to events & actions.

State model describes the control structure of objects. The interaction model focuses on the exchanges b/w objects & provides a holistic overview of the operation of the System.

CH 3. CLASS MODELING

Object & Class Concepts

An Object is a concept, abstraction, or thing with identity that has meaning for an application. Some objects have real-world counterparts while others are conceptual entities.

The choice of objects depends on judgment & the nature of a problem, there can be many correct representations.

Eg: Joe Smith, Simplex Company

Classes:

An object is an instance - or occurrence of a class. A class describes a group of objects with the same properties, behaviour, kinds of relationships & semantics.

Person, company, process are all classes.

Object in a class have the same attributes & form of behaviour. Most objects derive their individuality from differences in their attribute values & specific relationships to other objects.

We can write operations once for each class, so that all the objects in the class benefit from code reuse.

Class diagrams

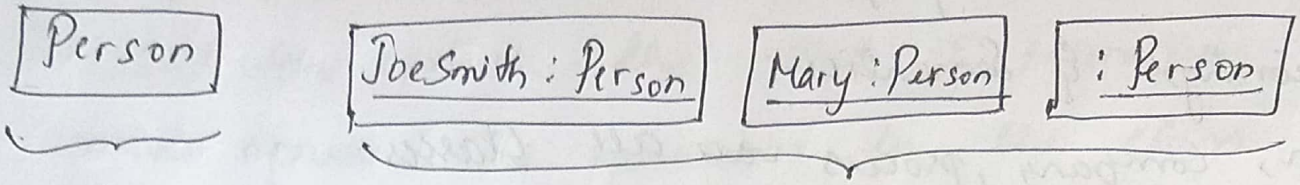
There are two kinds of models of structure - class diagram & object diagrams.

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

Object diagram shows individual objects & their relationships. Object diagrams are helpful for documenting test cases & and discussing

examples. A class diagram corresponds to an infinite set of object diagrams.

Eg:



Class Objects.

The above example shows the class & its instances. The UML symbol for an object is a box with an object name followed by a colon & the class name. Both are underlined. class name is also given as box symbol

Values & 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.

Name, birthdate & weight are attributes of Person object. Color, modelyear & weight are attributes of Car object.

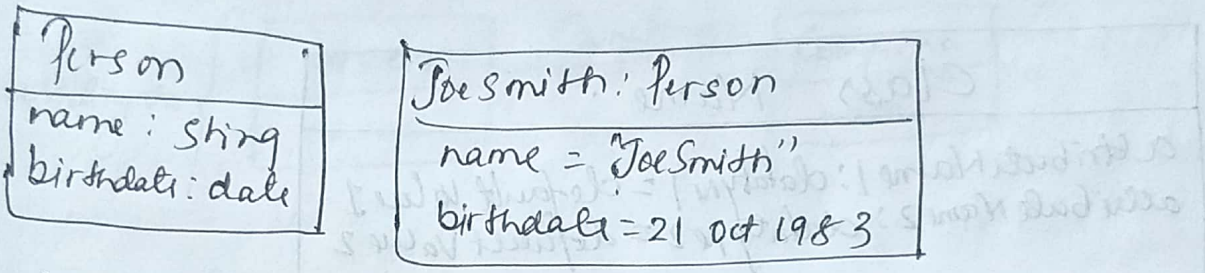
Each attribute has a value for each object.

Eg: Attribute birthdate has value "21 Oct 1983" for object JoeSmith.

The attributes values described values, not objects.

Following shows modelling notation. Class Person has attributes name & birthdate. Name is a String & birthdate is date.

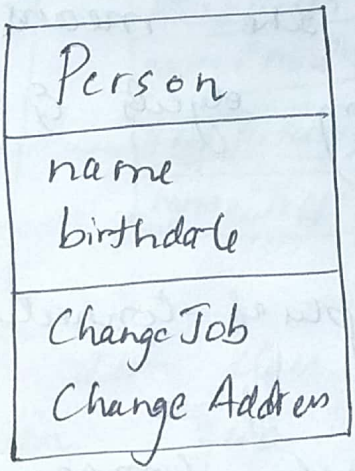
One person in class Person has the value "Joe Smith" for name & the value "21 oct 1983" for birthdate.



Class with Attributes. object with values

Operations & Methods

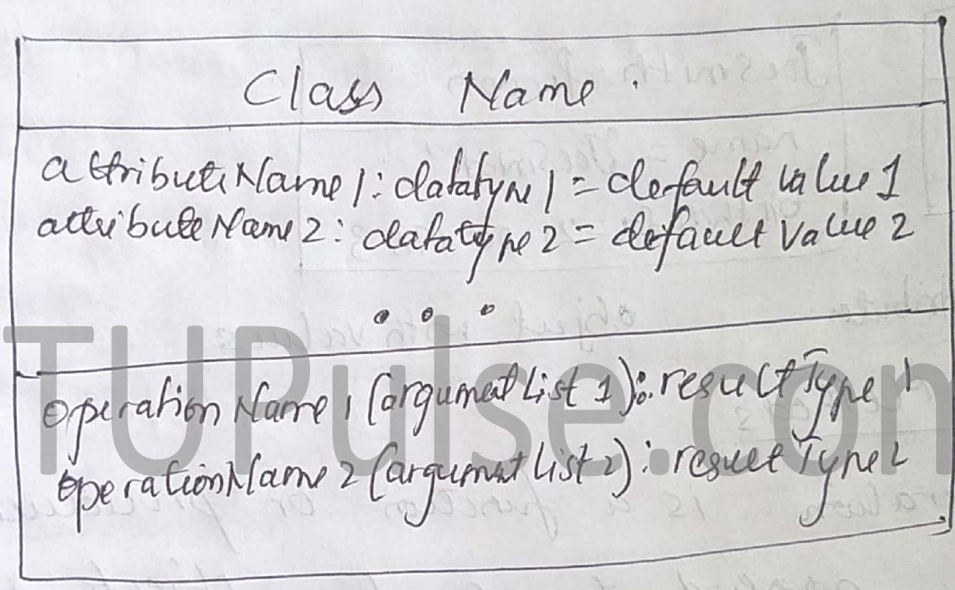
An operation is a function or procedure that may be applied to or by objects in a class. A method is the implementation of an operation for a class.



Eg:

Summary of Notation for classes

A box represents a class and may have as many as three compartments. The compartments contain from top to bottom, class name, list of attributes and list of operations.



* Link & Association Concepts

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

Link is a physical or conceptual connection among objects

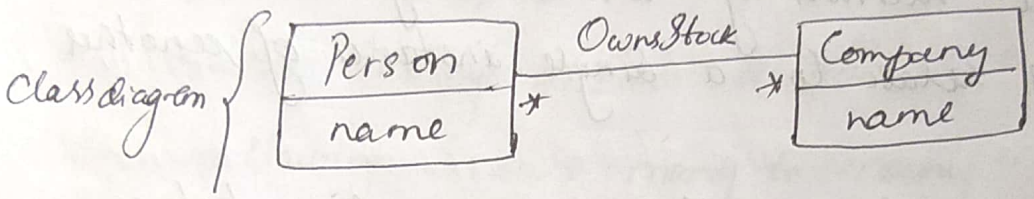
Eg: Joe Smith works for Simplex company

Most links relate two objects, but some

links relate three or more objects.

An association is a description of a group of links with common structure & common semantics. The link of association connect objects from the same classes.

Following example for class diagram & object diagram



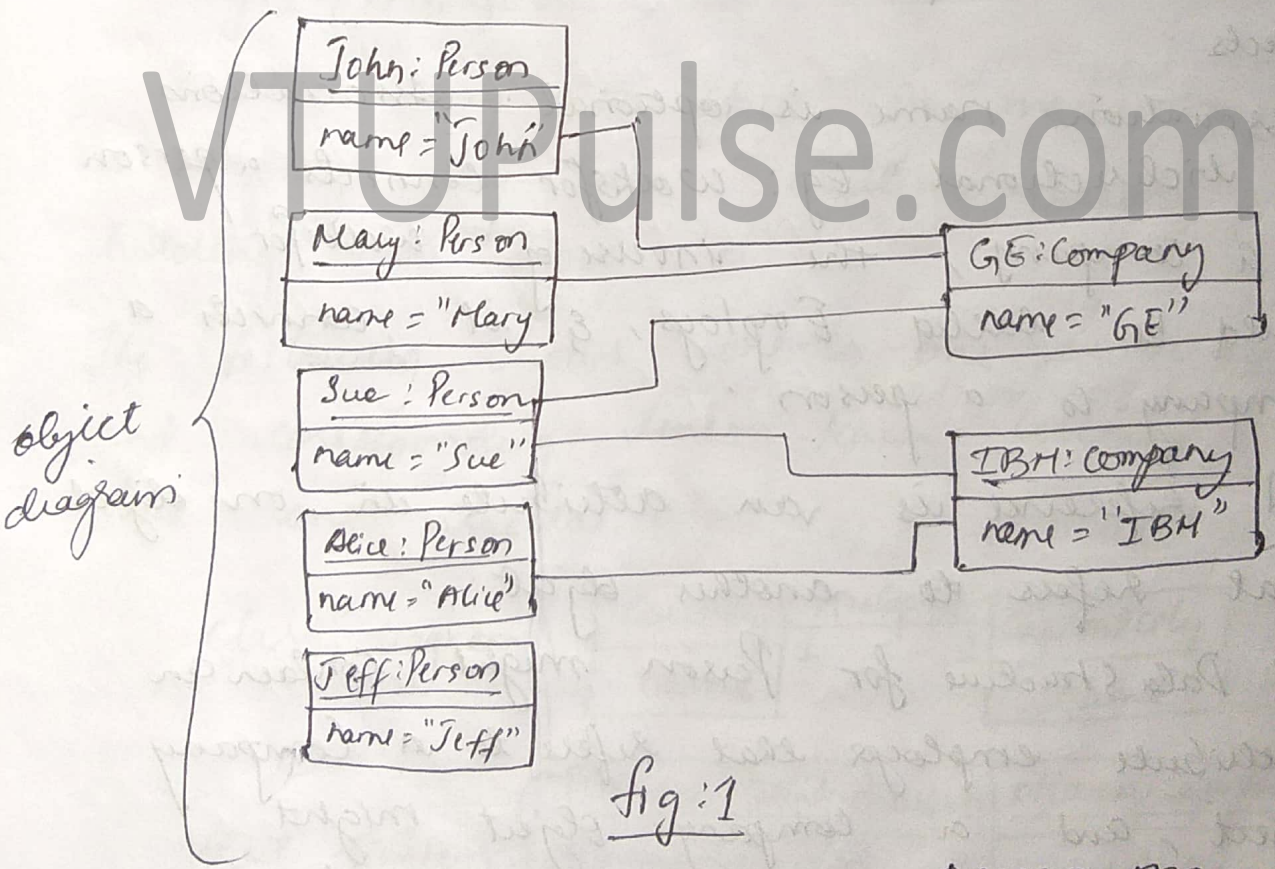


fig: 1

In 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, May & Sue own stock in GE company. Sue & Alice own stock in the IBM Company. Jeff doesnot own stock in any company and thus has no link.

→ Asterisk is a multiplicity symbol. Multiplicity specifies the number of instances of one class that may relate to a single instance of another class.

→ UML notation for a link is a line between objects.

→ Association name is optional. Associations are bidirectional. Eg: worksfor connects a person to a company, the inverse of worksfor could be called Employs, & it connects a company to a person.

→ A reference is an attribute in on object that refers to another object.
Eg: Data structure for Person might contain an attribute employs that refers to a Company object, and a company object might contain an attribute employees that refer to a list of Person objects.

Multiplicity: Specifies the number of instances of one class that may relate to a single instance of an associated class.

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

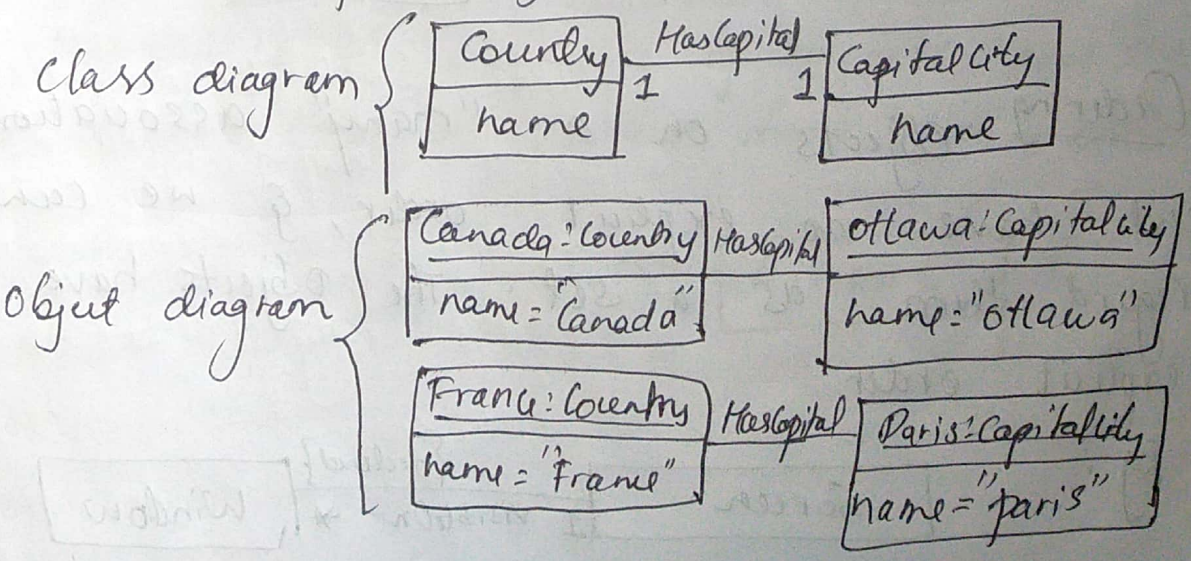
UML Specifies the intervals such as "1" (exactly one) "1..*" (one or more), or "3..5" (three to five)

* denotes "many" (Zero or more).

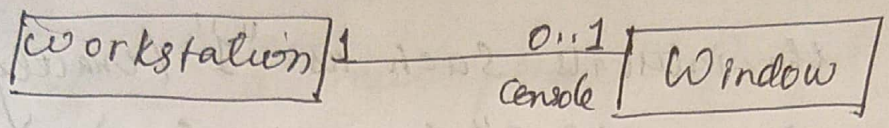
The following shows many to many multiplicity fig. 1. (refer)

A person may own stock in many companies, a company may have multiple persons holding its stock.

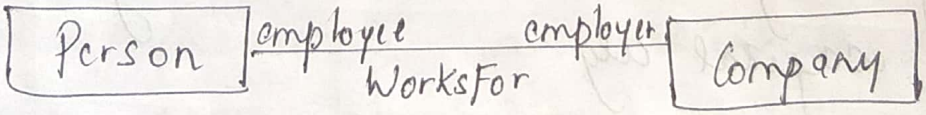
The following shows one-to-one association and corresponding links. Each country has one capital city.



Following illustrate 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.

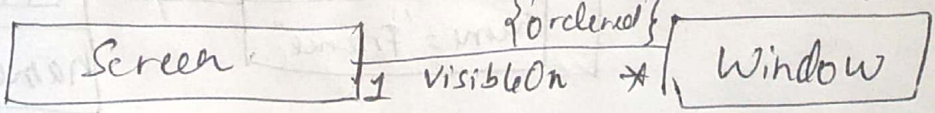


Association End Names: Each end of association can have a name. Association end names often appear as nouns in problem description. Following shows a name appears next to the association end. Person and Company participate in association WorksFor.
A person is an employee with respect to a ~~person~~ Company, Company is an employer with respect to a person.



Ordering: Objects on a "many" association end have no explicit order, & we can regard them as a set. The objects have explicit order.

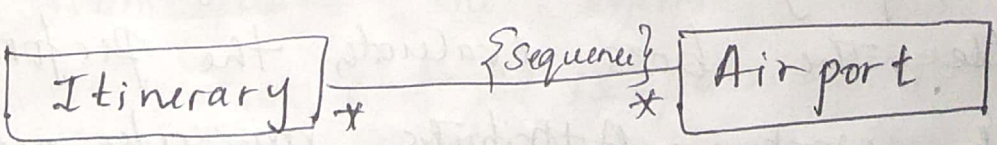
Eg:



The above example shows a workstation screen containing a number of overlapping windows. Each window on a screen occurs at most once. The windows have an explicit order, so only the top-most window is visible at any point on the screen.
The ordering is an inherent part of the association. We can indicate an ordered set of objects by writing "{ordered}" next to association end.

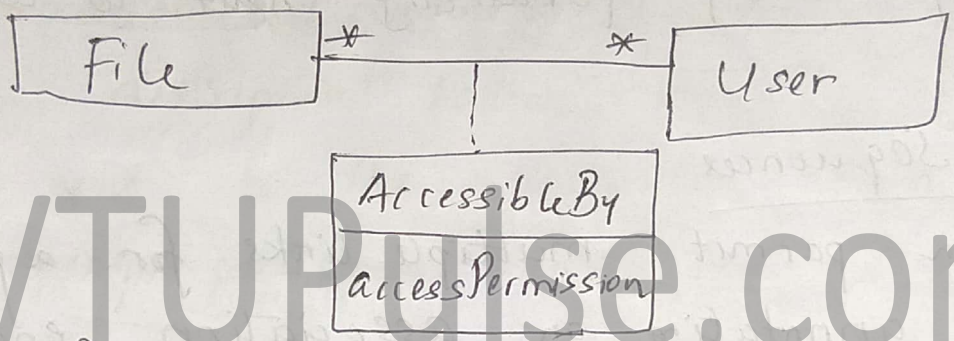
Bags & Sequences

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 duplicate allowed.
Eg: Itinerary is a sequence of airports and the same airport can be visited more than once



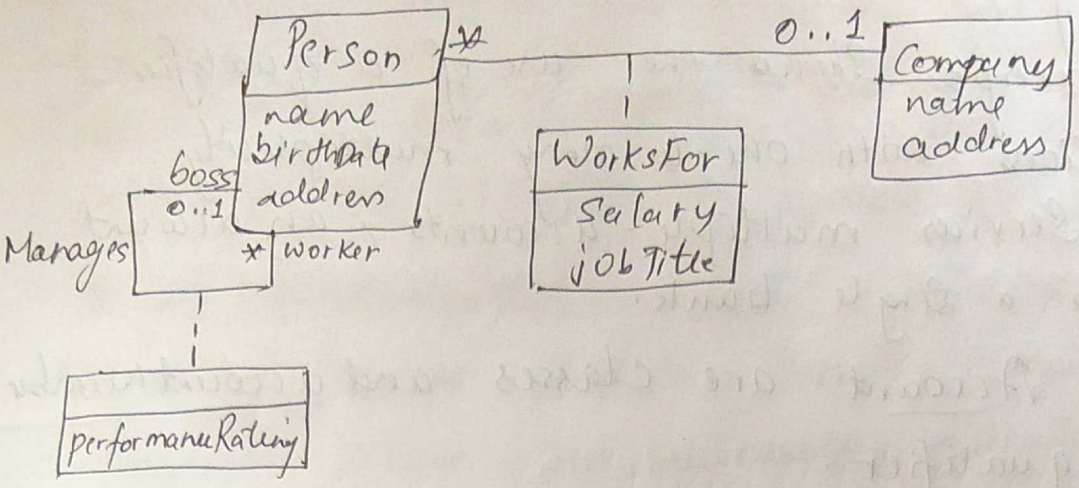
Association Classes

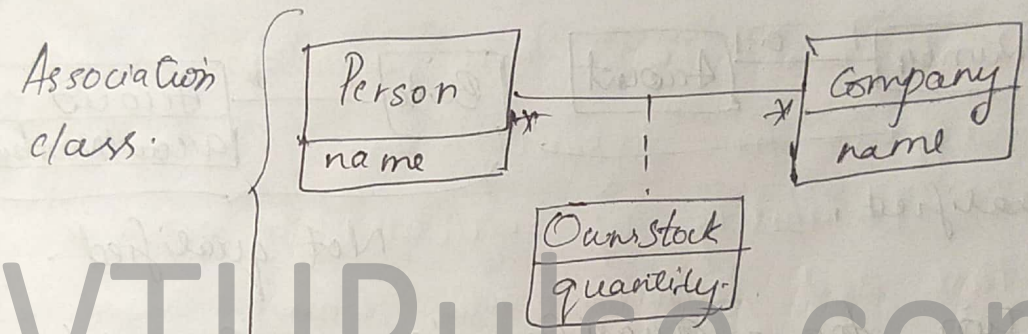
We can describe the links of an association with attributes. The UML represents such information with an association class.
It is an association that is also a class. Like the links of an association, the instances of an association class derive identity from instances of the constituent classes.

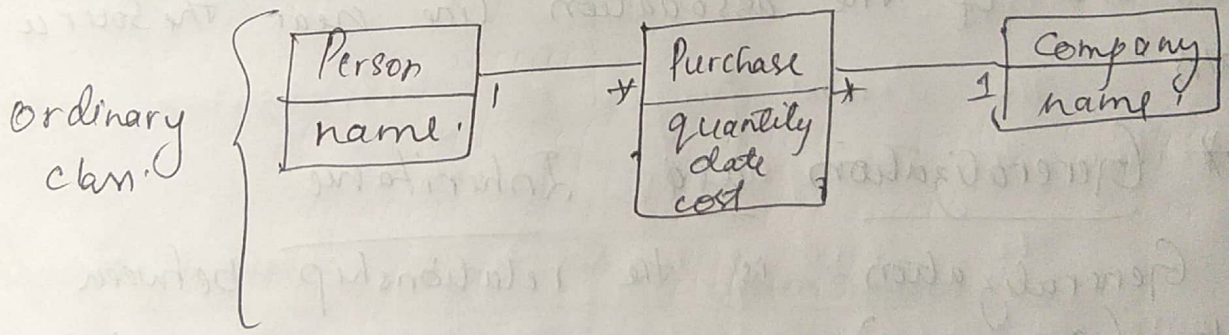


→ accessPermission is an attribute of AccessibleBy. The UML notation for an association class is a box attached to the association by a dashed line.

→ The following presents 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. Attributes may also occur for one-to-one associations.







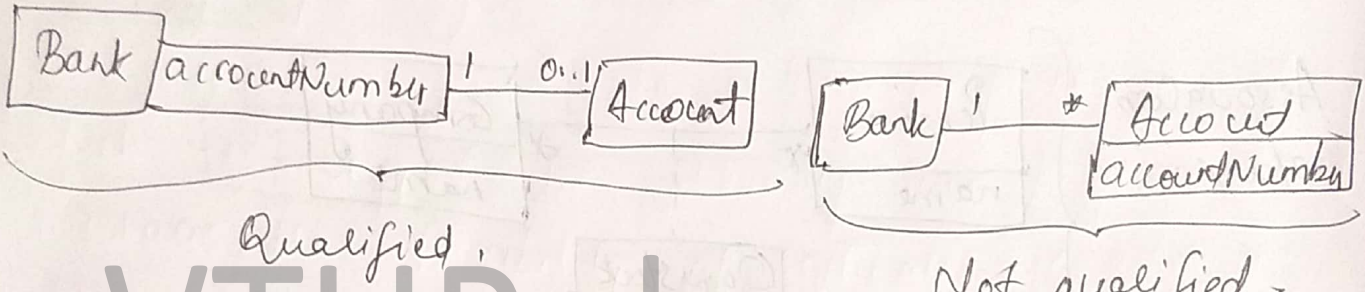
Qualified Associations: is an association in which an attribute called the qualifier disambiguates the objects for a "many" association end. It is possible to define qualifiers for one-to many and many to many associations. A qualifier selects among the target objects, reducing the effective multiplicity.

from "many" to "one".

Bellow example shows the use of a qualifier - for associations with one-to-many multiplicity.

A Bank services multiple accounts; an account belongs to a single bank.

Bank & Account are classes and accountNumber is the qualifier.



Qualified.

Not qualified.

The notation for a qualifier is a small box on the end of the association line near the source class.

* Generalization and Inheritance

Generalization is the relationship between a class (Superclass) and one or more variations of the class (Subclass). Generalization organizes classes by their similarities & differences, structuring the description of objects.

The Superclass holds common attributes, operations and associations, the Subclass add specific attributes, operations & associations.

Each Subclass is said to inherit the

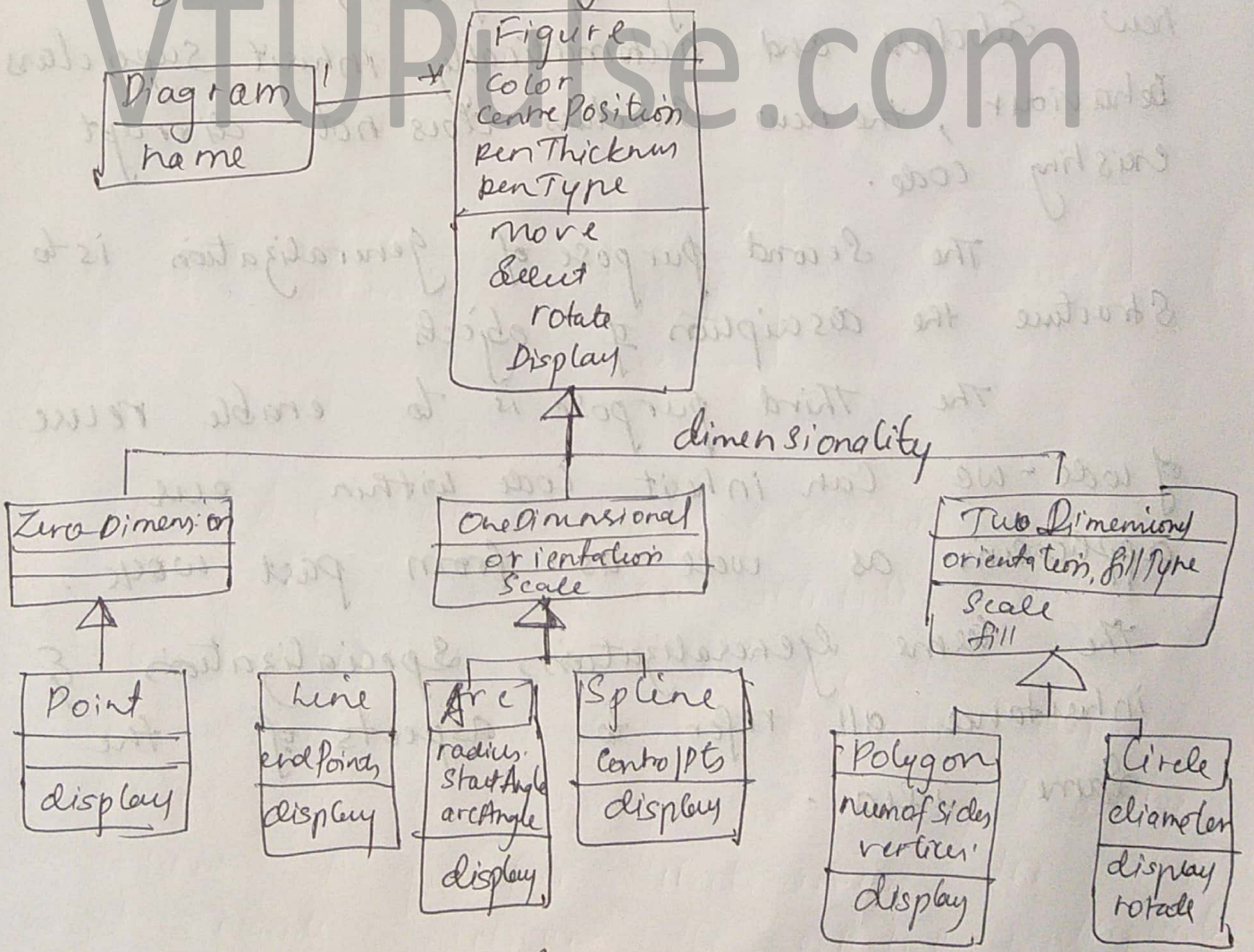
features of its superclass.

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

Single generalization organizes classes into a hierarchy, each subclass has a single immediate superclass. There can be multiple levels of generalizations.

The large hollow arrowhead denotes generalization. The arrowhead points to the superclass.

Following shows classes of geometric figures.



Inheritance for graphic figures.

The word written next to the generalization line in the diagram - 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.

Use of Generalization:

Generalization has three purposes, one of which is support for polymorphism. Polymorphism increases the flexibility of software, we add a new subclass and automatically inherit superclass behaviour, the new subclass does not disrupt existing code.

The second purpose of generalization is to structure the description of objects

The third purpose is to enable reuse of code - we can inherit code within our application as well as from past work.

The terms generalization, specialization & inheritance all refer to aspects of the same idea.

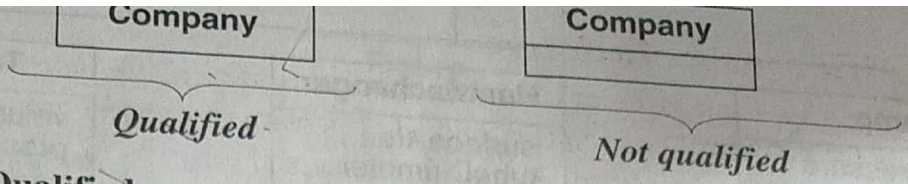


Figure 3.23 Qualified association. Qualification also facilitates traversal of class models.

3.3 Generalization and Inheritance

3.3.1 Definition

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 a hierarchy; each subclass has a single immediate superclass. (Chapter 4 discusses a more complex form of generalization in which a subclass may have multiple immediate superclasses.) There can be multiple levels of generalizations.

Figure 3.24 shows several examples of generalization for equipment. Each piece of equipment is a pump, heat exchanger, or tank. There are several kinds of pumps: centrifugal, diaphragm, and plunger. There are several kinds of tanks: spherical, pressurized, and 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. Thus *P101* embodies the features of equipment, pump, and diaphragm pump. *E302* has the properties of equipment and heat exchanger.

A large hollow arrowhead denotes generalization. The arrowhead points to the superclass. You may directly connect the superclass to each subclass, but we normally prefer to group subclasses as a tree. For convenience, you can rotate the triangle and place it on any side, but if possible you should draw the superclass on top and the subclasses on the bottom. The curly braces denote a UML comment, indicating that there are additional subclasses that the diagram does not show.

A Model is an Obstruction of Something for the purpose of understanding it before building it

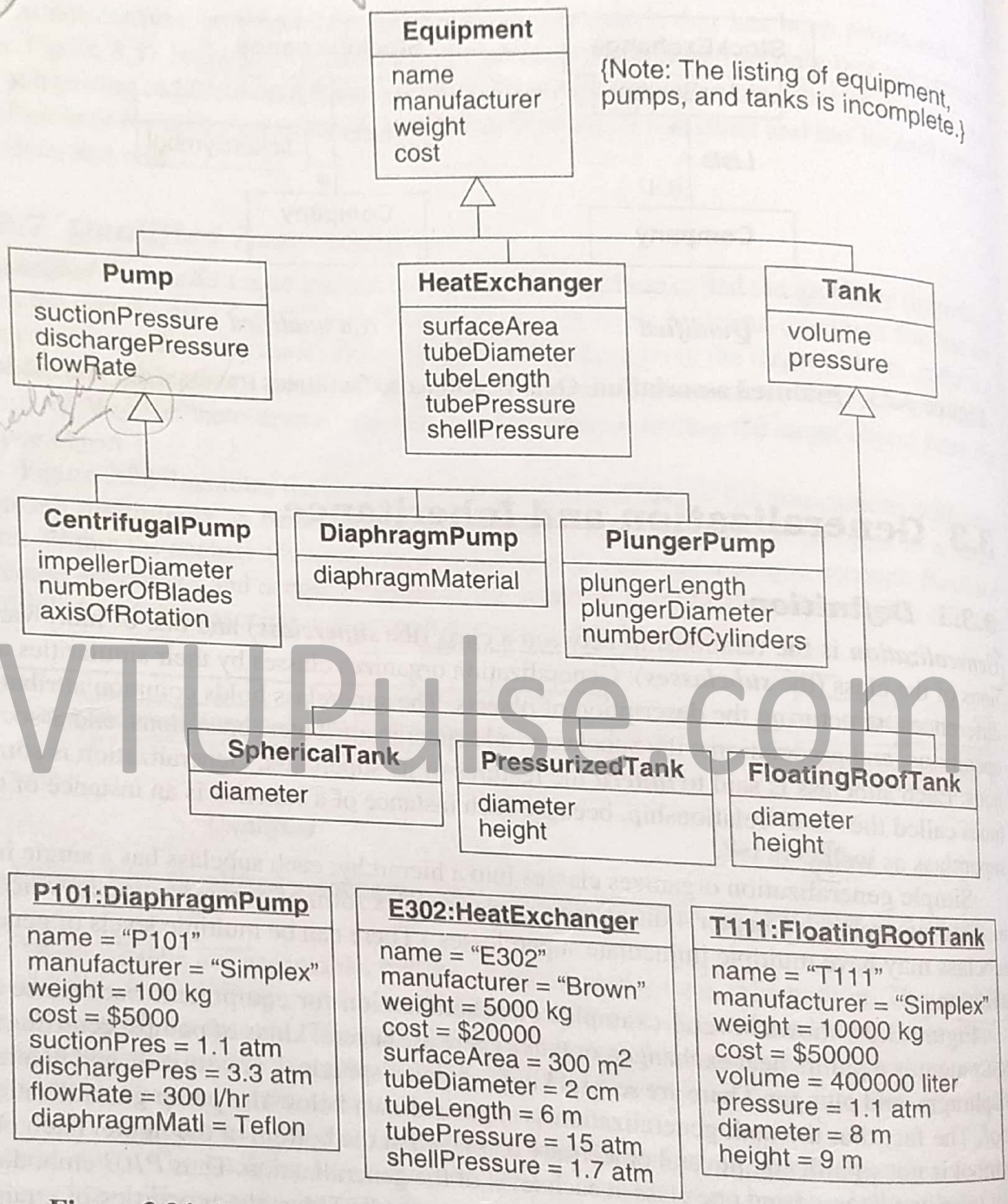


Figure 3.24 A multilevel inheritance hierarchy with instances. Generalization organizes classes by their similarities and differences, structuring the description of objects.

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. 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. For example, *Pump* adds attributes *suctionPressure*, *dischargePressure*, and *flowRate*, which other kinds of equipment do not share.

Figure 3.25 shows classes of geometric figures. This example has more of a programming flavor and emphasizes inheritance of operations. *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.

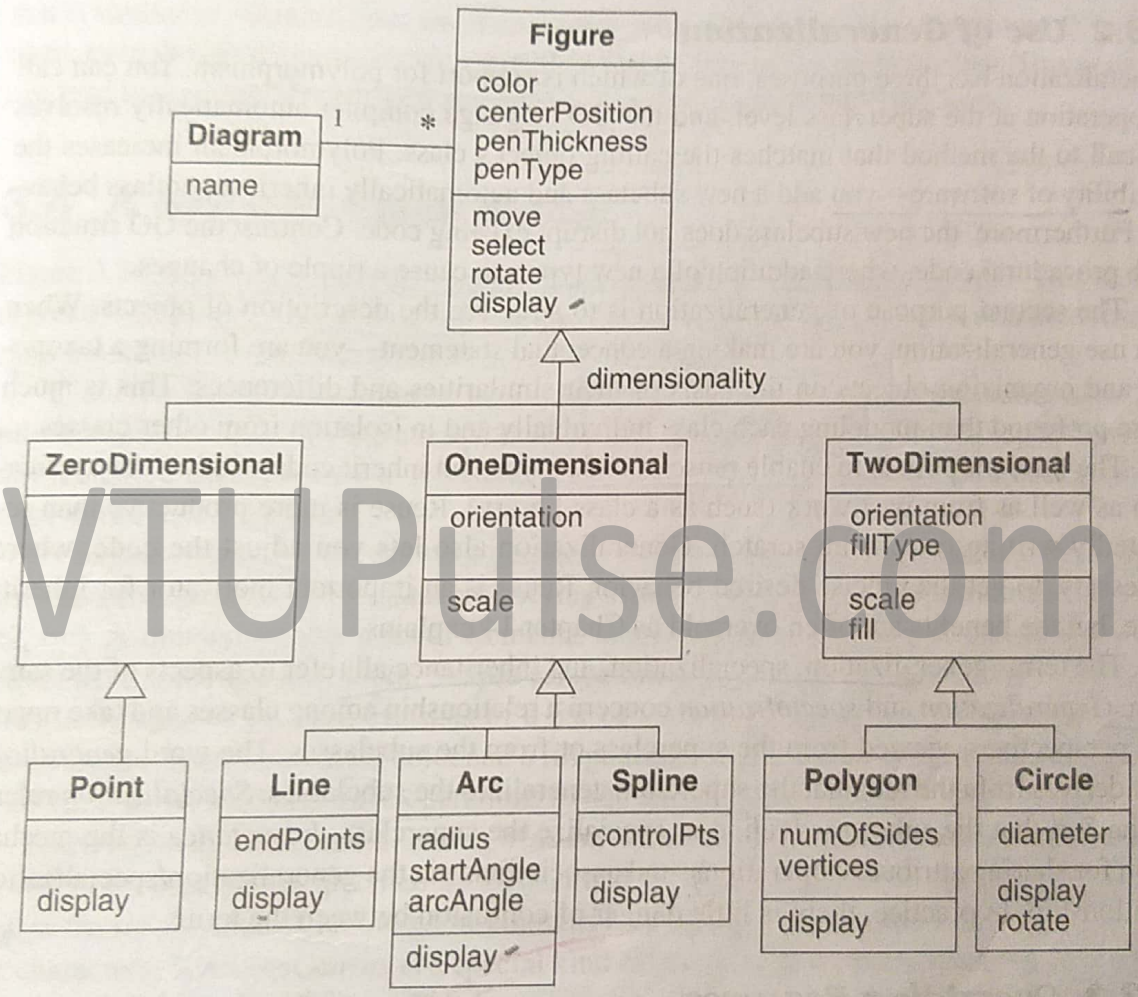


Figure 3.25 Inheritance for graphic figures. Each subclass inherits the attributes, operations, and associations of its superclasses.

The word written next to the generalization line in the diagram—*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. You 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.

with a special method that takes advantage of specific information but does not alter the operation semantics (such as *Circle.rotate* in Figure 3.25).

You 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 common, but unfortunate, practice in OO programming is to "borrow" a class that is similar to a desired class and then modify it by changing and ignoring some of its features, even though the new class is not really a special case of the original class. This practice can lead to conceptual confusion and hidden assumptions built into programs.

3.4 A Sample Class Model

Figure 3.26 shows a class model of a workstation window management system. This model is greatly simplified—a real model would require a number of pages—but it illustrates many class modeling constructs and shows how they fit together.

Class *Window* defines common parameters of all kinds of windows, including a rectangular boundary defined by the attributes *x1, y1, x2, y2*, and operations to display and undisplay a window and to raise it to the top (foreground) or lower it to the bottom (background) of the 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 *cx1, cy1, cx2, cy2*. 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 scrolling; it is both a *Canvas* and a *ScrollingWindow*. This is an example of *multiple inheritance*, to be explained in Chapter 4.

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 three kinds: buttons, choice items, and 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 two associations between *ChoiceItem* and *ChoiceEntry*; a one-to-many as-

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

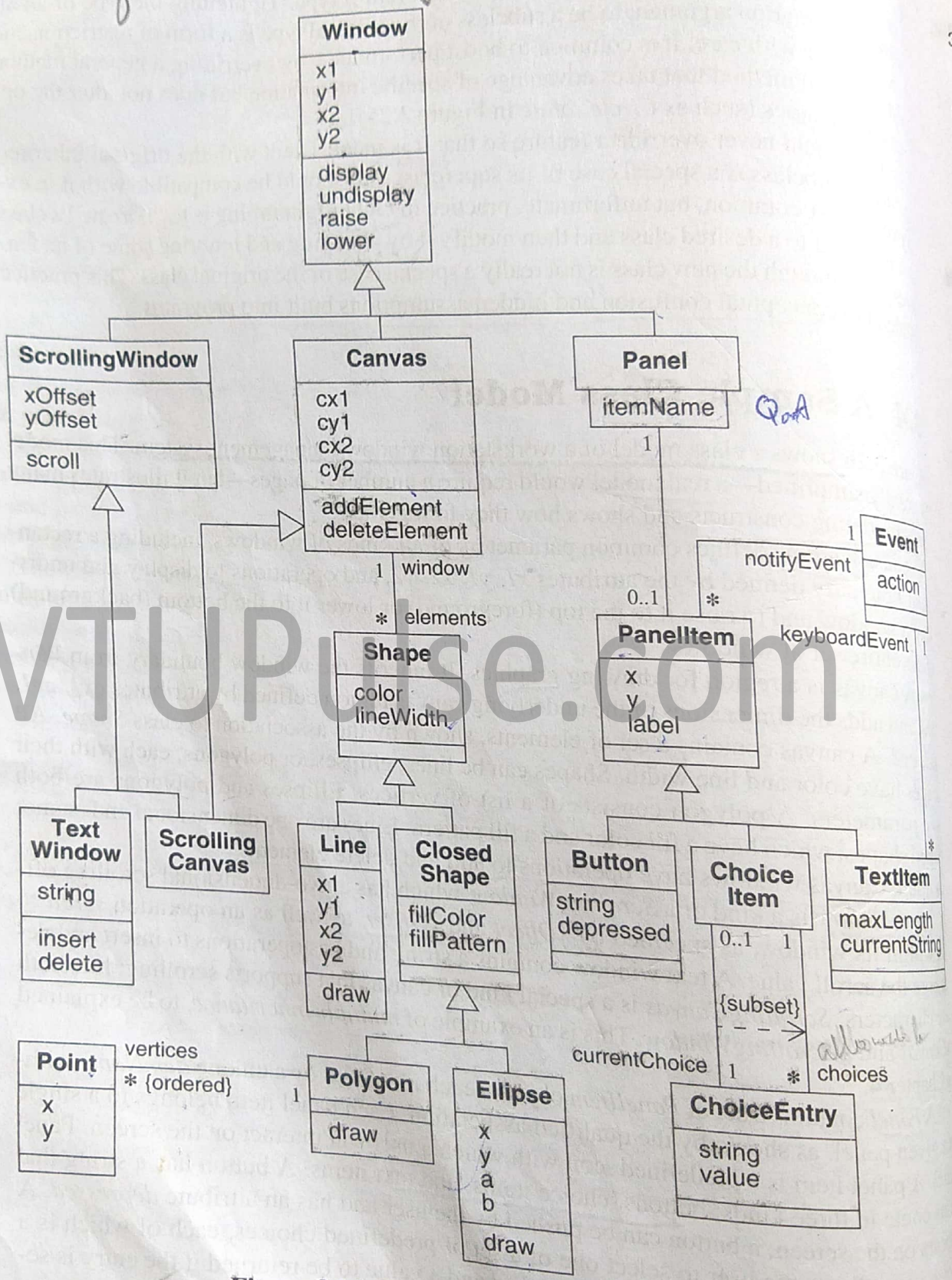


Figure 3.26 Class model of a windowing system

Workstation windowing management systems

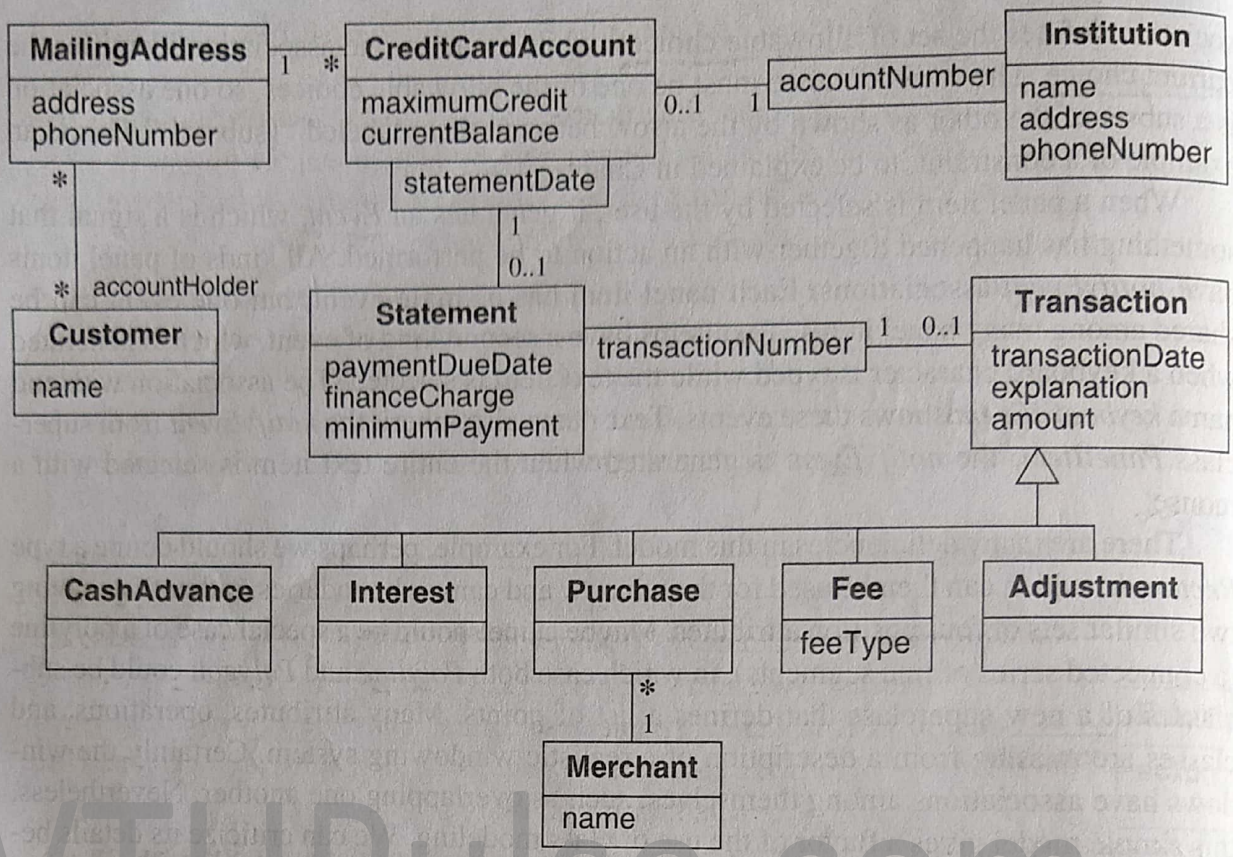


Figure 3.27 Class model for managing credit card accounts

- 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)*** [Warmer-99]. The next two sections discuss the OCL, and Section 3.5.3 then expresses the credit card questions using the OCL. By no means do we cover the complete OCL; we just cover the portions relevant to traversing class models.

3.5.1 OCL Constructs for Traversing Class Models

The OCL can traverse the constructs in class models.

- **Attributes**. You can traverse from an object to an attribute value. The syntax is the source object, followed by a dot, and then the attribute name. For example, the expression *aCreditCardAccount.maximumCredit* takes a *CreditCardAccount* object and finds the value of *maximumCredit*. (We use the convention of preceding a class name by "a" to refer to an object.) Similarly, you can access an attribute for each object in a collection, returning a collection of attribute values. In addition, you can find an attribute value for a link, or a collection of attribute values for a collection of links.
- **Operations**. You 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, and then the operation. An operation must be followed by parentheses, even if it has no arguments, to

avoid confusion with attributes. You may invoke operations from your class model or predefined operations that are built into the OCL.

The OCL has special operations that operate on entire collections (as opposed to operating on each object in a collection). For example, you can count the objects in a collection or sum a collection of numeric values. The syntax for a collection operation is the source object collection, followed by "->", and then the operation.

- **Simple associations**. A third use of the *dot* notation is 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. In the example, *aCustomer.MailingAddress* yields a set of addresses for a customer (the target end has "many" multiplicity). In contrast, *aCreditCardAccount.MailingAddress* yields a single address (the target end has multiplicity of one).
- **Qualified associations**. A qualifier lets you make a more precise traversal. The expression *aCreditCardAccount.Statement[30 November 1999]* finds the statement for a credit card account with the statement date of 30 November 1999. The syntax is to enclose the qualifier value in brackets. Alternatively, you 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 not used.)
- **Association classes**. Given a link of an association class, you can find the constituent objects. Alternatively, given a constituent object, you 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. The *select* operation applies a predicate to each element in a collection and returns the elements that satisfy the predicate. For example, *aStatement.Transaction->select(amount>$100)* finds the transactions for a statement in excess of $100.