3

Class Modeling

A class model captures the static structure of a system by characterizing the objects in the system, the relationships between the objects, and the attributes and operations for each class of objects. The class model is the most important of the three models. We emphasize building a system around objects rather than around functionality, because an object-oriented system more closely corresponds to the real world and is consequently more resilient with respect to change. Class models provide an intuitive graphic representation of a system and are valuable for communicating with customers.

Chapter 3 discusses basic class modeling concepts that will be used throughout the book. We define each concept, present the corresponding UML notation, and provide examples. Some important concepts that we consider are object, class, link, association, generalization, and inheritance. You should master the material in this chapter before proceeding in the book.

3.1 Object and Class Concepts

3.1.1 Objects

The purpose of class modeling is to describe objects. For example, *Joe Smith*, *Simplex company*, *process number 7648*, and *the top window* are objects.

An ***object*** is a concept, abstraction, or thing with identity that has meaning for an application. Objects often appear as proper nouns or specific references in problem descriptions and discussions with users. Some objects have real-world counterparts (Albert Einstein and the General Electric company), while others are conceptual entities (simulation run 1234 and the formula for solving a quadratic equation). Still others (binary tree 634 and the array bound to variable *a*) are introduced for implementation reasons and have no correspondence to physical reality. The choice of objects depends on judgment and the nature of a problem; there can be many correct representations.

All objects have identity and are distinguishable. Two apples with the same color, shape, and texture are still individual apples; a person can eat one and then eat the other. Similarly, identical twins are two distinct persons, even though they may look the same. The term ***identity*** means that objects are distinguished by their inherent existence and not by descriptive properties that they may have.

3.1.2 Classes

An object is an ***instance***—or occurrence—of a class. A ***class*** describes a group of objects with the same properties (attributes), behavior (operations), kinds of relationships, and semantics. *Person*, *company*, *process*, and *window* are all classes. Each person has name and birthdate and may work at a job. Each process has an owner, priority, and list of required resources. Classes often appear as common nouns and noun phrases in problem descriptions and discussions with users.

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. However, objects with identical attribute values and relationships are possible. The choice of classes depends on the nature and scope of an application and is a matter of judgment.

The objects in a class share a common semantic purpose, above and beyond the requirement of common attributes and behavior. For example, a barn and a horse may both have a cost and an age. If barn and horse were regarded as purely financial assets, they could belong to the same class. If the developer took into consideration that a person paints a barn and feeds a horse, they would be modeled as distinct classes. 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.

If objects are the focus of modeling, why bother with classes? The notion of abstraction is at the heart of the matter. By grouping objects into classes, we abstract a problem. Abstraction gives modeling its power and ability to generalize from a few specific cases to a host of similar cases. Common definitions (such as class name and attribute names) are stored once per class rather than once per instance. You can write operations once for each class, so that all the objects in the class benefit from code reuse. For example, all ellipses share the same procedures to draw them, compute their areas, and test for intersection with a line; polygons would have a separate set of procedures. Even special cases, such as circles and squares, can use the general procedures, though more efficient procedures are possible.

3.1.3 Class Diagrams

We began this chapter by discussing some basic modeling concepts, specifically *object* and *class*. We have described these concepts with examples and prose. This approach is vague and insufficient for dealing with the complexity of applications. We need a means for expressing models that is coherent, precise, and easy to formulate. There are two kinds of models of structure—class diagrams and 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. They are concise, easy to understand, and work well in practice. We will use class diagrams throughout this book to represent the structure of applications.

We will also occasionally use object diagrams. 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 shows a class (left) and instances (right) described by it. Objects *JoeSmith*, *MarySharp*, and 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. Our convention is to list the object name and class name in boldface.

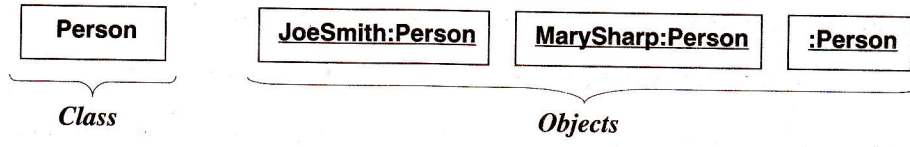


Figure 3.1 A class and objects. Objects and classes are the focus of class modeling.

The UML symbol for a class also is a box. Our convention is to list the class name in boldface, center the name in the box, and capitalize the first letter. We use singular nouns for the names of classes.

Note how we run together multiword names, such as *JoeSmith*, separating the words with intervening capital letters. This is the convention we use for referring to objects, classes, and other constructs. Alternative conventions would be to use intervening spaces (Joe Smith) or underscores (Joe_Smith). The mixed capitalization convention is popular in the OO literature but is not a UML requirement.

3.1.4 Values and Attributes

A ***value*** is a piece of data. You can find values by examining problem documentation for examples. An ***attribute*** is a named property of a class that describes a value held by each object of the class. You can find attributes by looking for adjectives or by abstracting typical values. The following analogy holds: Object is to class as value is to attribute. Structural constructs—that is, classes and relationships (to be explained)—dominate class models. Attributes are of lesser importance and serve to elaborate classes and relationships.

Name, *birthdate*, and *weight* are attributes of *Person* objects. *Color*, *modelYear*, and *weight* are attributes of *Car* objects. Each attribute has a value for each object. For example, attribute *birthdate* has value "21 October 1983" for object *JoeSmith*. Paraphrasing, Joe Smith was born on 21 October 1983. Different objects may have the same or different values for a given attribute. Each attribute name is unique within a class (as opposed to being unique across all classes). Thus class *Person* and class *Car* may each have an attribute called *weight*.

Do not confuse values with objects. An attribute should describe values, not objects. Unlike objects, values lack identity. For example, all occurrences of the integer "17" are indistinguishable, as are all occurrences of the string "Canada." The country Canada is an object, whose *name* attribute has the value "Canada" (the string).

Figure 3.2 shows modeling notation. Class *Person* has attributes *name* and *birthdate*. *Name* is a 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 and the value "16 March 1950" for birthdate.

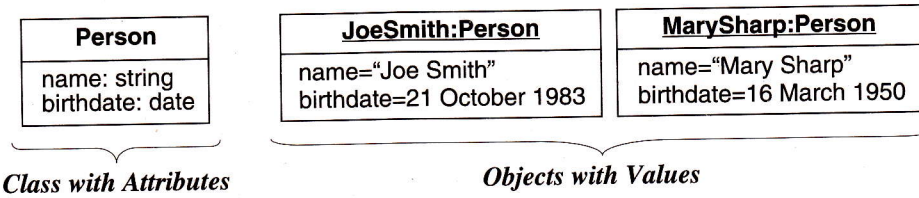


Figure 3.2 Attributes and values. Attributes elaborate classes.

The UML notation lists attributes in the second compartment of the class box. Optional details, such as type and default value, may follow each attribute. A colon precedes the type. An equal sign precedes the default value. Our convention is to show the attribute name in regular face, left align the name in the box, and use a lowercase letter for the first letter.

You may also include attribute values in the second compartment of object boxes. The notation is to list each attribute name followed by an equal sign and the value. We also left align attribute values and use regular type face.

Some implementation media require that an object have a unique identifier. These identifiers are implicit in a class model—you need not and should not list them explicitly. Figure 3.3 emphasizes the point. Most OO languages automatically generate identifiers with which to reference objects. You can also readily define them for databases. Identifiers are a computer artifact and have no intrinsic meaning.

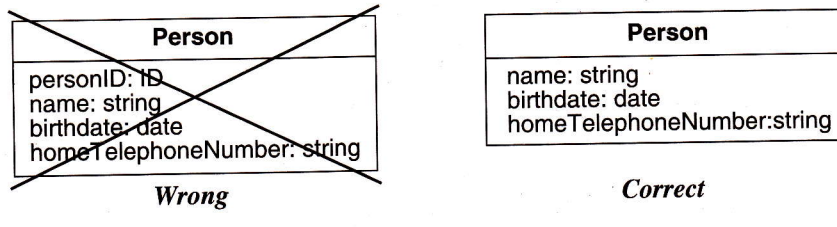


Figure 3.3 Object identifiers. Do not list object identifiers; they are implicit in models.

Do not confuse internal identifiers with real-world attributes. Internal identifiers are purely an implementation convenience and have no application meaning. In contrast, tax payer number, license plate number, and telephone number are not internal identifiers because they have meaning in the real world. Rather they are legitimate attributes.

3.1.5 Operations and Methods

An ***operation*** is a function or procedure that may be applied to or by objects in a class. *Hire*, *fire*, and *payDividend* are operations on class *Company*. *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 behavior of the operation depends on the class of its target. An object "knows" its class, and hence the right implementation of the operation.

The same operation may apply to many different classes. Such an operation is ***polymorphic;*** that is, the same operation takes on different forms in different classes. A ***method*** is the implementation of an operation for a class. For example, the class *File* may have an operation *print*. You could implement different methods to print ASCII files, print binary files, and print digitized picture files. All these methods logically perform the same task—printing a file; thus you may refer to them by the generic operation *print*. However, a different piece of code may implement each method.

An operation may have arguments in addition to its target object. Such arguments may be placeholders for values, or for other objects. The choice of a method depends entirely on the class of the target object and not on any object arguments that an operation may have. (A few OO languages, notably CLOS, permit the choice of method to depend on any number of arguments, but such generality leads to considerable semantic complexity; which we shall not explore.)

When an operation has methods on several classes, it is important that the methods all have the same ***signature***—the number and types of arguments and the type of result value. For example, *print* should not have *fileName* as an argument for one method and *filePointer* for another. The behavior of all methods for an operation should have a consistent intent. It is best to avoid using the same name for two operations that are semantically different, even if they apply to distinct sets of classes. For example, it would be unwise to use the name *invert* to describe both a matrix inversion and turning a geometric figure upside-down. In a large project, some form of name scoping may be necessary to accommodate accidental name clashes, but it is best to avoid any possibility of confusion.

In Figure 3.4, the class *Person* has attributes *name* and *birthdate* and operations *changeJob* and *changeAddress*. *Name*, *birthdate*, *changeJob*, and *changeAddress* are features of *Person*. ***Feature*** is a generic word for either an attribute or operation. Similarly, *File* has a *print* operation. *GeometricObject* has *move*, *select*, and *rotate* operations. *Move* has argument *delta,* which is a *Vector*; *select* has one argument *p*, which is of type *Point* and returns a *Boolean*; and *rotate* has argument *angle*, which is an input of type float with a default value of 0.0.

The UML notation is to list operations in the third compartment of the class box. Our convention is 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 parentheses shows explicitly that there are no arguments; otherwise you cannot draw conclusions. We do not list operations for objects, because they do not vary among objects of the same class.

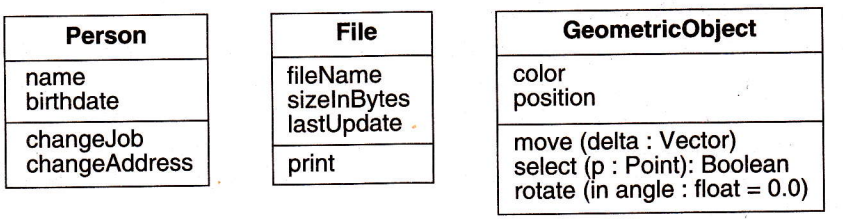


Figure 3.4 Operations. An operation is a function or procedure that may be applied to or by objects in a class.

3.1.6 Summary of Notation for Classes

Figure 3.5 summarizes the 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. Optional details such as type and default value may follow each attribute name. Optional details such as argument list and result type may follow each operation name.

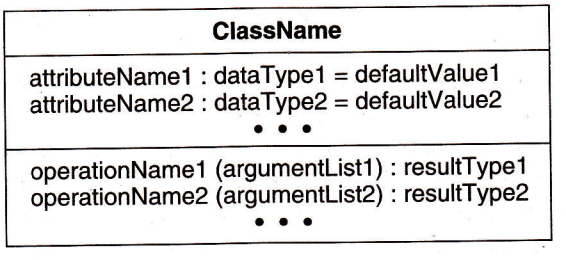


Figure 3.5 Summary of modeling notation for classes. A box represents a class and may have as many as three compartments.

Figure 3.6 shows that each argument may have a direction, name, type, and default value. The ***direction*** indicates whether an argument is an input (*in*), output (*out*), or an input argument that can be modified (*inout*). A colon precedes the type. An equal sign precedes the default value. The default value is used if no argument is supplied for the argument.

direction argumentName : type = defaultValue

Figure 3.6 Notation for an argument of an operation. The direction, type, and default value are optional. Direction may be *in*, *out*, or *inout*.

The attribute and operation compartments of class boxes are optional, and you may or may not show them. A missing attribute compartment means that attributes are unspecified. Similarly, a missing operation compartment means that operations are unspecified. In contrast, an empty compartment means that attributes (operations) are specified and that there are none.

3.2 Link and Association Concepts

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

3.2.1 Links and Associations

A ***link*** is a physical or conceptual connection among objects. For example, Joe Smith *WorksFor* Simplex company. Most links relate two objects, but some links relate three or more objects. This chapter discusses only binary associations; Chapter 4 discusses n-ary associations. Mathematically, we define a link as a tuple—that is, a list of objects. A link is an instance of an association.

An ***association*** is a description of a group of links with common structure and common semantics. For example, 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 statements.

Figure 3.7 is an excerpt of a model for a financial application. Stock brokerage firms need to perform tasks such as recording ownership of various stocks, tracking dividends, alerting customers to changes in the market, and computing margin requirements. The top portion of the figure shows a class diagram and the bottom shows an object diagram.

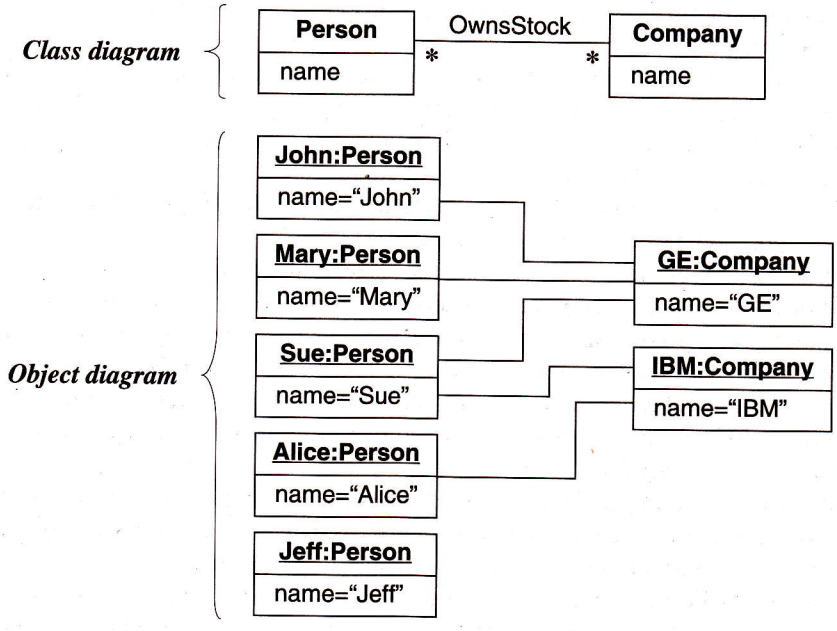


Figure 3.7 Many-to-many association. An association describes a set of potential links in the same way that a class describes a set of potential objects.

In the class diagram, a person may own stock in zero or more companies; a company may have multiple persons owning its stock. The object diagram shows some examples. John, Mary, and Sue own stock in the GE company. Sue and Alice own stock in the IBM company. Jeff does not own stock in any company and thus has no link. The asterisk is a multiplicity symbol. Multiplicity specifies the number of instances of one class that may relate to a single instance of another class and is discussed in the next section.

The UML notation for a link is a line between objects; a line may consist of several line segments. If the link has a name, it is underlined. For example, John owns stock in the GE company. An association connects related classes and is also denoted by a line (with possibly multiple line segments). For example, persons own stock in companies. Our convention is to show link and association names in italics and to confine line segments to a rectilinear grid. It is good to arrange the classes in an association to read from left-to-right, if possible.

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* and *person owns stock in company*). When there are multiple associations, you must use association names or association end names (Section 3.2.3) to resolve the ambiguity.

Associations are inherently bidirectional. The name of a binary association usually reads in a particular direction, but the binary association can be traversed in either direction. For example, *WorksFor* connects a person to a company. The inverse of *WorksFor* could be called *Employs,* and it connects a company to a person. In reality, both directions of traversal are equally meaningful and refer to the same underlying association; it is only the names that establish a direction.

Developers often implement associations in programming languages as references from one object to another. A ***reference*** is an attribute in one object that refers to another object. For example, a data structure for *Person* might contain an attribute *employer* that refers to a *Company* object, and a *Company* object might contain an attribute *employees* that refers to a set of *Person* objects. Implementing associations as references is perfectly acceptable, but you should not model associations this way.

A link is a relationship among objects. Modeling a link as a reference disguises the fact that the link is not part of either object by itself, but depends on both of them together. A company is not part of a person, and a person is not part of a company. Furthermore, using a pair of matched references, such as the reference from *Person* to *Company* and the reference from *Company* to a set of *Persons,* hides the fact that the forward and inverse references depend on each other. Therefore, you should model all connections among classes as associations, even in designs for programs.

The OO literature emphasizes encapsulation, that implementation details should be kept private to a class, and we certainly agree with this. Associations are important, precisely because they break encapsulation. Associations cannot be private to a class, because they transcend classes. Failure to treat associations on an equal footing with classes can lead to programs containing hidden assumptions and dependencies. Such programs are difficult to extend and the classes are difficult to reuse.

Although modeling treats associations as bidirectional, you do not have to implement them in both directions. You can readily implement associations as references if they are only

traversed in a single direction. Chapter 17 discusses some trade-offs to consider when implementing associations.

3.2.2 Multiplicity

Multiplicity specifies the number of instances of one class that may relate to a single instance of an associated class. Multiplicity constrains the number of related objects. The literature often describes multiplicity as being "one" or "many," but more generally it is a (possibly infinite) subset of the nonnegative integers. 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, inclusive). The special symbol "*" is a shorthand notation that denotes "many" (zero or more).

Figure 3.7 illustrates many-to-many multiplicity. A person may own stock in many companies. A company may have multiple persons holding its stock. In this particular case, John and Mary own stock in the GE company; Alice owns stock in the IBM company; Sue owns stock in both companies; Jeff does not own any stock. GE stock is owned by three persons; IBM stock is owned by two persons.

Figure 3.8 shows a one-to-one association and some corresponding links. Each country has one capital city. A capital city administers one country. (In fact, some countries, such as The Netherlands and Switzerland, have more than one capital city for different purposes. If this fact were important, the model could be modified by changing the multiplicity or by providing a separate association for each kind of capital city.)

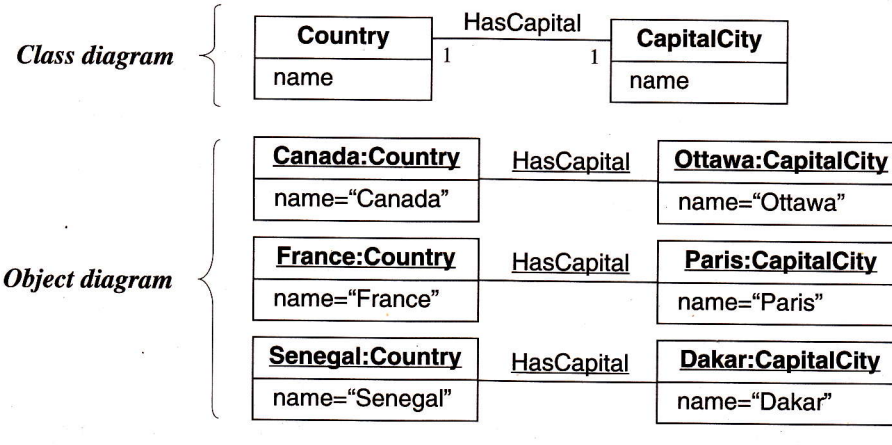


Figure 3.8 One-to-one association. Multiplicity specifies the number of instances of one class that may relate to a single instance of an associated class.

Figure 3.9 illustrates zero-or-one multiplicity. A workstation may have one of its windows designated as the console to receive general error messages. It is possible, however, that no console window exists. (The word "console" on the diagram is an association end name, discussed in Section 3.2.3.)

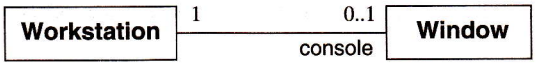


Figure 3.9 Zero-or-one multiplicity. It may be optional whether an object is involved in an association.

Do not confuse "multiplicity" with "cardinality." Multiplicity is a *constraint* on the size of a collection; cardinality is the *count* of elements that are actually in a collection. Therefore, multiplicity is a constraint on the cardinality.

A multiplicity of "many" specifies that an object may be associated with multiple objects. However, for each association there is at most one link between a given pair of objects (except for bags and sequences, see Section 3.2.5). As Figure 3.10 and Figure 3.11 show, if you want two links between the same objects, you must have two associations.

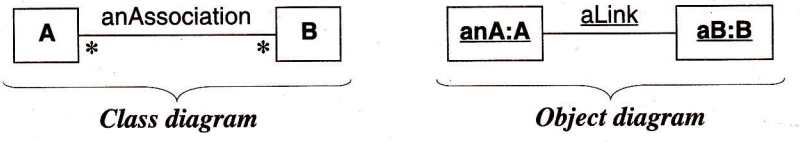


Figure 3.10 Association vs. link. A pair of objects can be instantiated at most once per association (except for bags and sequences).

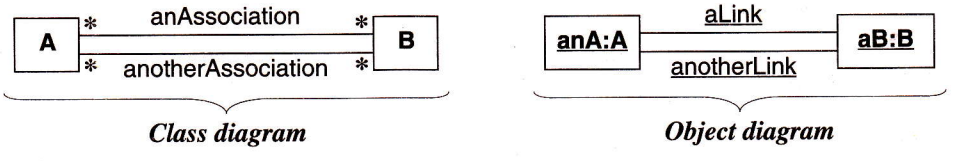


Figure 3.11 Association vs. link. You can use multiple associations to model multiple links between the same objects.

Multiplicity depends on assumptions and how you define the boundaries of a problem. Vague requirements often make multiplicity uncertain. Do not worry excessively about multiplicity early in software development. First determine classes and associations, then decide on multiplicity. If you omit multiplicity notation from a diagram, multiplicity is considered to be unspecified.

Multiplicity often exposes hidden assumptions built into a model. For example, is the *WorksFor* association between *Person* and *Company* one-to-many or many-to-many? It depends on the context. A tax collection application would permit a person to work for multiple companies. On the other hand, the member records for an auto workers' union may consider second jobs irrelevant. Class diagrams help to elicit these hidden assumptions, making them visible and subject to scrutiny.

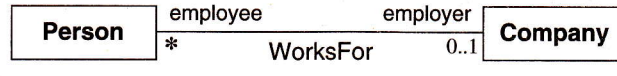
The most important multiplicity distinction is between "one" and "many." Underestimating multiplicity can restrict the flexibility of an application. For example, many programs

cannot accommodate persons with multiple phone numbers. On the other hand, overestimating multiplicity imposes overhead and requires the application to supply additional information to distinguish among the members of a "many" set. In a true hierarchical organization, for example, it is better to represent "boss" with a multiplicity of "zero or one," rather than allow for nonexistent matrix management.

3.2.3 Association End Names

Our discussion of multiplicity implicitly referred to the ends of associations. For example, a one-to-many association has two ends—an end with a multiplicity of "one" and an end with a multiplicity of "many." The notion of an ***association end*** is an important concept in the UML. You can not only assign a multiplicity to an association end, but you can give it a name as well. (Chapter 4 discusses additional properties of association ends.)

Association end names often appear as nouns in problem descriptions. As Figure 3.12 shows, a name appears next to the association end. In the figure *Person* and *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.



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

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

Association end names are especially convenient for traversing associations, because you can treat each one as a pseudo attribute. Each end of a binary association refers to an object or set of objects associated with a source object. From the point of view of the source object, traversal of the association is an operation that yields related objects. Association end names provide a means of traversing an association, without explicitly mentioning the association. Section 3.5 talks further about traversing class models.

Association end names are necessary for associations between two objects of the same class. For example, in Figure 3.13 *container* and *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 between the same pair of classes. In Figure 3.13 each directory has exactly one user who is an owner and many users who are authorized to use the directory. When there is only a single association between a pair of distinct classes, the names of the classes often suffice, and you may omit association end names.

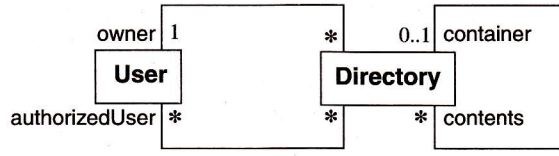


Figure 3.13 Association end names. Association end names are necessary for associations between two objects of the same class. They can also distinguish multiple associations between a pair of classes.

Association end names let you unify multiple references to the same class. When constructing class diagrams you should properly use association end names and not introduce a separate class for each reference, as Figure 3.14 shows. In the wrong model, two instances represent a person with a child, one for the child and one for the parent. In the correct model, one person instance participates in two or more links, twice as a parent and zero or more times as a child. (In the correct model, we must show a child as having an optional parent, so that the recursion eventually terminates.)

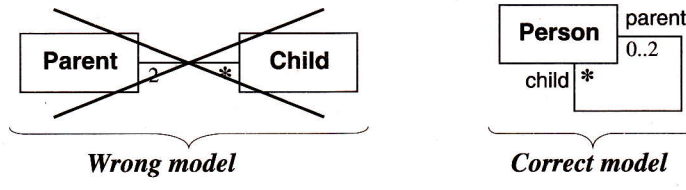


Figure 3.14 Association end names. Use association end names to model multiple references to the same class.

Because association end names distinguish objects, all names on the far end of associations attached to a class must be unique. Although the name appears next to the destination object on an association, it is really a pseudo attribute of the source class and must be unique within it. For the same reason, no association end name should be the same as an attribute name of the source class.

3.2.4 Ordering

Often the objects on a "many" association end have no explicit order, and you can regard them as a set. Sometimes, however, the objects have an explicit order. For example, Figure 3.15 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 topmost window is visible at any point on the screen. The ordering is an inherent part of the association. You can indicate an ordered set of objects by writing "{ordered}" next to the appropriate association end.

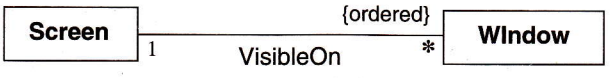


Figure 3.15 Ordering the objects for an association end. Ordering sometimes occurs for "many" multiplicity.

3.2.5 Bags and Sequences

Ordinarily a binary association has at most one link for a pair of objects. However, you can permit multiple links for a pair of objects by annotating an association end with *{bag}* or *{sequence}*. A ***bag*** is a collection of elements with duplicates allowed. A ***sequence*** is an ordered collection of elements with duplicates allowed. In Figure 3.16 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.

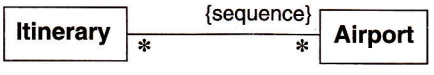


Figure 3.16 An example of a sequence. An itinerary may visit multiple airports, so you should use *{sequence}* and not *{ordered}*.

UML1 did not permit multiple links for a pair of objects. Some modelers misunderstood this restriction with ordered association ends and constructed incorrect models, assuming that there could be multiple links. With UML2 the modeler's intent is now clear. If you specify *{bag}* or *{sequence}*, then there can be multiple links for a pair of objects. If you omit these annotations, then the association has at most one link for a pair of objects.

Note that the *{ordered}* and the *{sequence}* annotations are the same, except that the first disallows duplicates and the other allows them. A sequence association is an ordered bag, while an ordered association is an ordered set.

3.2.6 Association Classes

Just as you can describe the objects of a class with attributes, so too you can describe the links of an association with attributes. The UML represents such information with an association class. 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. You can find association classes by looking for adverbs in a problem statement or by abstracting known values.

In Figure 3.17, *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.

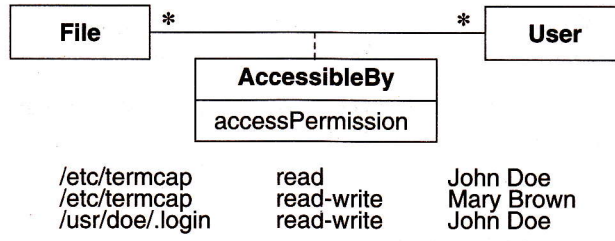


Figure 3.17 An association class. The links of an association can have attributes.

Many-to-many associations provide a compelling rationale for association classes. Attributes for such associations unmistakably belong to the link and cannot be ascribed to either object. In Figure 3.17, *accessPermission* is a joint property of *File* and *User* and cannot be attached to either *File* or *User* alone without losing information.

Figure 3.18 presents attributes for two one-to-many associations. Each person working for a company receives a salary and has a job title. The boss evaluates the performance of each worker. Attributes may also occur for one-to-one associations.

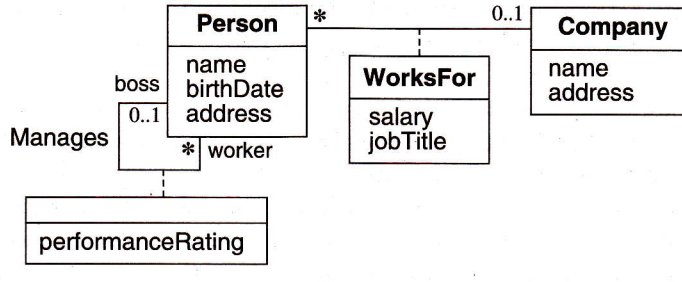


Figure 3.18 Association classes. Attributes may also occur for one-to-many and one-to-one associations.

Figure 3.19 shows how it is possible to fold attributes for one-to-one and one-to-many associations into the class opposite a "one" end. This is not possible for many-to-many associations. As a rule, you should not fold such attributes into a class because the multiplicity of the association might change. Either form in Figure 3.19 can express a one-to-many association. However, only the association class form remains correct if the multiplicity of *WorksFor* is changed to many-to-many.

Figure 3.20 shows an association class participating in an association. Users may be authorized on many workstations. Each authorization carries a priority and access privileges. A user has a home directory for each authorized workstation, but several workstations and users can share the same home directory. Association classes are an important aspect of class modeling because they let you specify identity and navigation paths precisely.

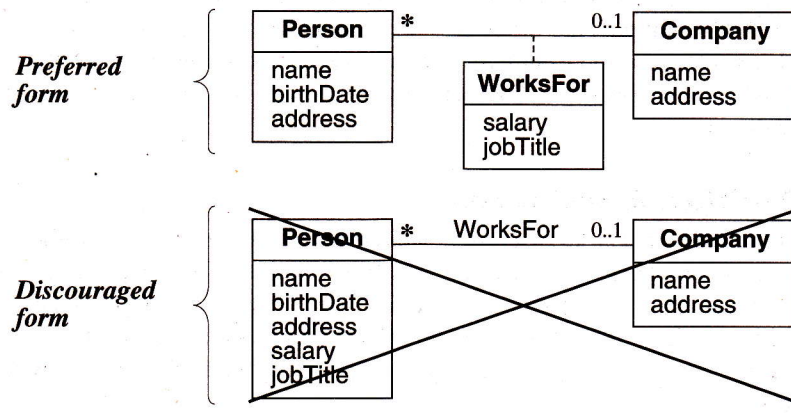


Figure 3.19 Proper use of association classes. Do not fold attributes of an association into a class.

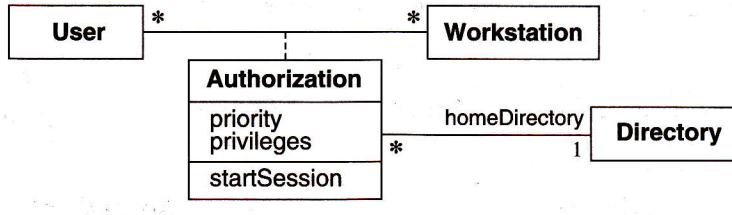


Figure 3.20 An association class participating in an association. Association classes let you specify identity and navigation paths precisely.

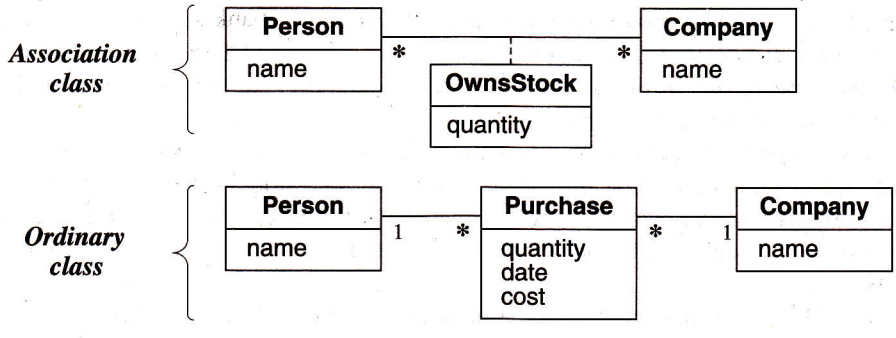


Figure 3.21 Association class vs. ordinary class. An association class is much different than an ordinary class.

Do not confuse an association class with an association that has been promoted to a class. Figure 3.21 highlights the difference. The association class has only one occurrence for each pairing of *Person* and *Company.* In contrast there can be any number of occurrences of a *Purchase* for each *Person* and *Company.* Each purchase is distinct and has its own quantity, date, and cost.

3.2.7 Qualified Associations

A ***qualified association*** is an association in which an attribute called the ***qualifier*** disambiguates the objects for a "many" association end. It is possible to define qualifiers for one-to-many and many-to-many associations. A qualifier selects among the target objects, reducing the effective multiplicity, from "many" to "one." 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 illustrates the most common use of a qualifier— for associations with one-to-many multiplicity. A bank services multiple accounts. An account belongs to a single bank. Within the context of a bank, the account number specifies a unique account. *Bank* and *Account* are classes and *accountNumber* is the qualifier. Qualification reduces the effective multiplicity of this association from one-to-many to one-to-one.

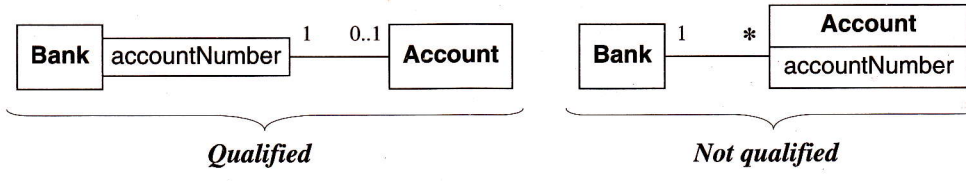


Figure 3.22 Qualified association. Qualification increases the precision of a model.

Both models are acceptable, but the qualified model adds information. The qualified model adds a multiplicity constraint, that the combination of a bank and an account number yields at most one account. The qualified model conveys the significance of account number in traversing the model, as methods will reflect. You first find the bank and then specify the account number to find the 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 Figure 3.22 *Bank* + *accountNumber* yields an *Account*, therefore *accountNumber* is listed in a box contiguous to *Bank.*

Figure 3.23 provides another example of qualification. 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. (We are presuming this is true. If every stock had a single ticker symbol that was invariant across exchanges, we would make *tickerSymbol* an attribute of *Company.*)

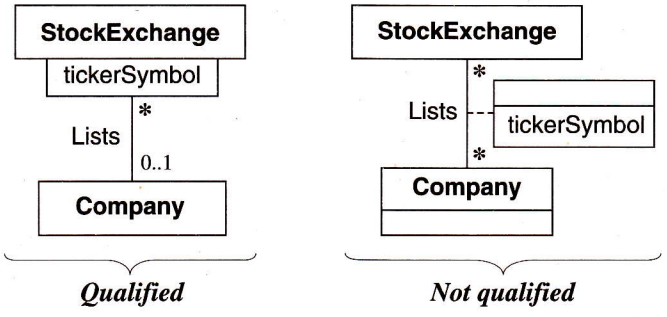


Figure 3.23 Qualified association. Qualification also facilitates traversal of class models.

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.

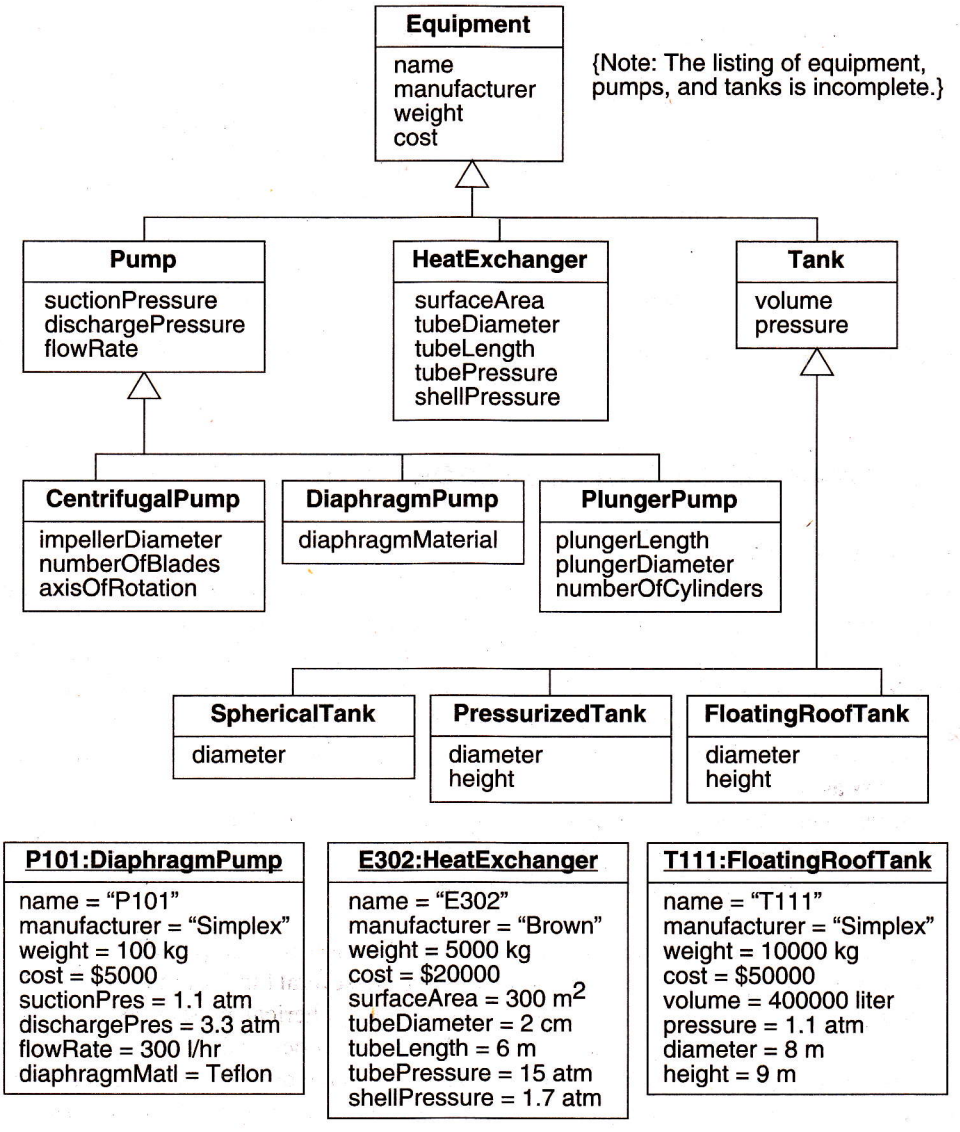


Figure 3.24 A multilevel inheritance hierarchy with instances. Generalization organizes classes by their similarities and differences, structuring the description of objects.

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.

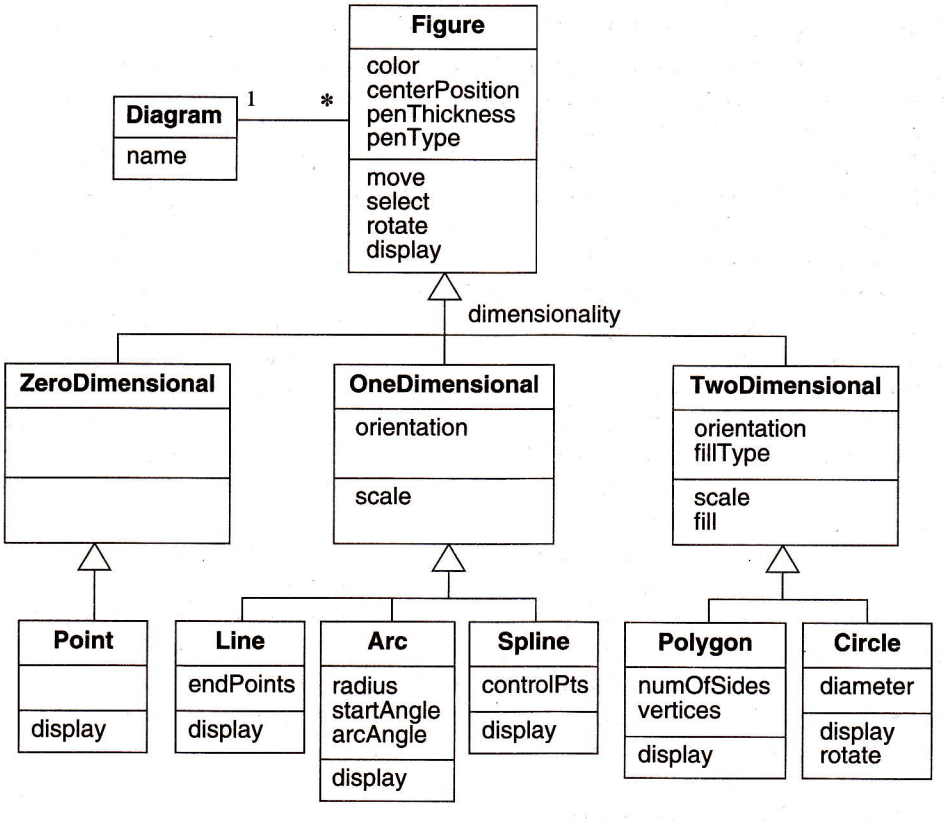


Figure 3.25 Inheritance for graphic figures. Each subclass inherits the attributes, operations, and associations of its superclasses.

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.

Do not nest subclasses too deeply. Deeply nested subclasses can be difficult to understand, much like deeply nested blocks of code in a procedural language. Often, with some careful thought and a little restructuring, you can reduce the depth of an overextended inheritance hierarchy. In practice, whether or not a subclass is "too deeply nested" depends upon judgment and the particular details of a problem. The following guidelines may help: 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.

3.3.2 Use of Generalization

Generalization has three purposes, one of which is support for polymorphism. You can call an operation at the superclass level, and the OO language compiler automatically resolves the call to the method that matches the calling object's class. Polymorphism increases the flexibility of software—you add a new subclass and automatically inherit superclass behavior. Furthermore, the new subclass does not disrupt existing code. Contrast the OO situation with procedural code, where addition of a new type can cause a ripple of changes.

The second purpose of generalization is to structure the description of objects. When you use generalization, you are making a conceptual statement—you are forming a taxonomy and organizing objects on the basis of their similarities and differences. This is much more profound than modeling each class individually and in isolation from other classes.

The third purpose is to enable reuse of code—you can inherit code within your 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 you adjust the code, where necessary, to get the precise desired behavior. Reuse is an important motivator for inheritance, but the benefits are often oversold as Chapter 14 explains.

The terms generalization, specialization, and inheritance all refer to aspects of the same idea. *Generalization* and *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, and associations via the generalization/specialization relationship. In practice, there is little danger of confusion between the terms.

3.3.3 Overriding Features

A subclass may ***override*** a superclass feature by defining a feature with the same name. The overriding feature (the subclass feature) refines and replaces the overridden feature (the superclass feature). There are several reasons why you may wish to override a feature: to specify behavior that depends on the subclass, to tighten the specification of a feature, or to improve performance. For example, in Figure 3.25, each leaf subclass must implement *display*, even though *Figure* defines it. Class *Circle* improves performance by overriding operation *rotate* to be a null operation.

You may override methods and default values of attributes. You should never override the *signature,* or form, of a feature. An override should preserve attribute type, number and

type of arguments to an operation, and operation return type. Tightening the type of an attribute or operation argument to be a subclass of the original type is a form of restriction and must be done with care. It is common to boost performance by overriding a general method 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-

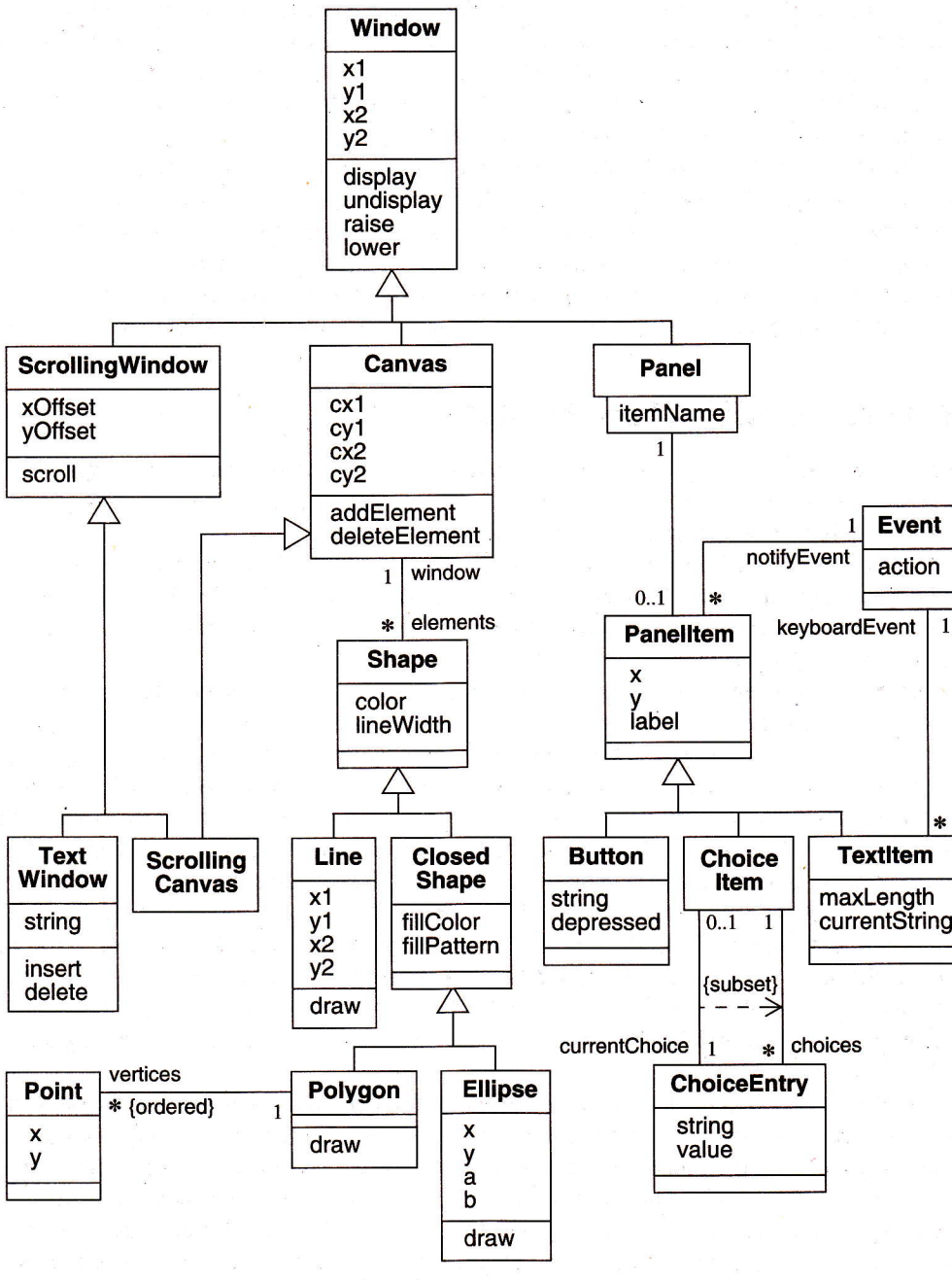


Figure 3.26 Class model of a windowing system

sociation 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 between them labeled "{subset}." This is an example of a constraint, to be explained in Chapter 4.

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. Text items also inherit the *notifyEvent* from superclass *PanelItem;* the *notifyEvent* is generated when the entire text item is selected with a mouse.

There are many deficiencies in this model. For example, perhaps we should define a type *Rectangle*, which can then be used for the window and canvas boundaries, rather than having two similar sets of four position attributes. Maybe a line should be a special case of a polyline (a connected series of line segments), in which case both *Polyline* and *Polygon* could be subclasses of a new superclass that defines a list of points. Many attributes, operations, and classes are missing from a description of a realistic windowing system. Certainly the windows have associations among themselves, such as overlapping one another. Nevertheless, this simple model gives a flavor of the use of class modeling. We can criticize its details because it says something precise. It would serve as the basis for a fuller model.

3.5 Navigation of Class Models

So far we have shown how class models can express the structure of an application. Now we show how they can also express the behavior of navigating among classes. Navigation is important because it lets you exercise a model and uncover hidden flaws and omissions so that you can repair them. You can perform navigation manually (an informal technique) or write navigation expressions (as we will explain).

Consider the simple model for credit card accounts in Figure 3.27. An institution may issue many credit card accounts, each identified by an account number. Each account has a maximum credit limit, a current balance, and 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 statement lists a payment due date, finance charge, and minimum payment. The statement itemizes various transactions that have occurred throughout the billing interval: cash advances, interest charges, purchases, fees, and adjustments to the account. The name of the merchant is printed for each purchase.

We can pose a variety of questions against the model.

- What transactions occurred for a credit card account within a time interval?
- What volume of transactions were handled by an institution in the last year?
- What customers patronized a merchant in the last year by any kind of credit card?

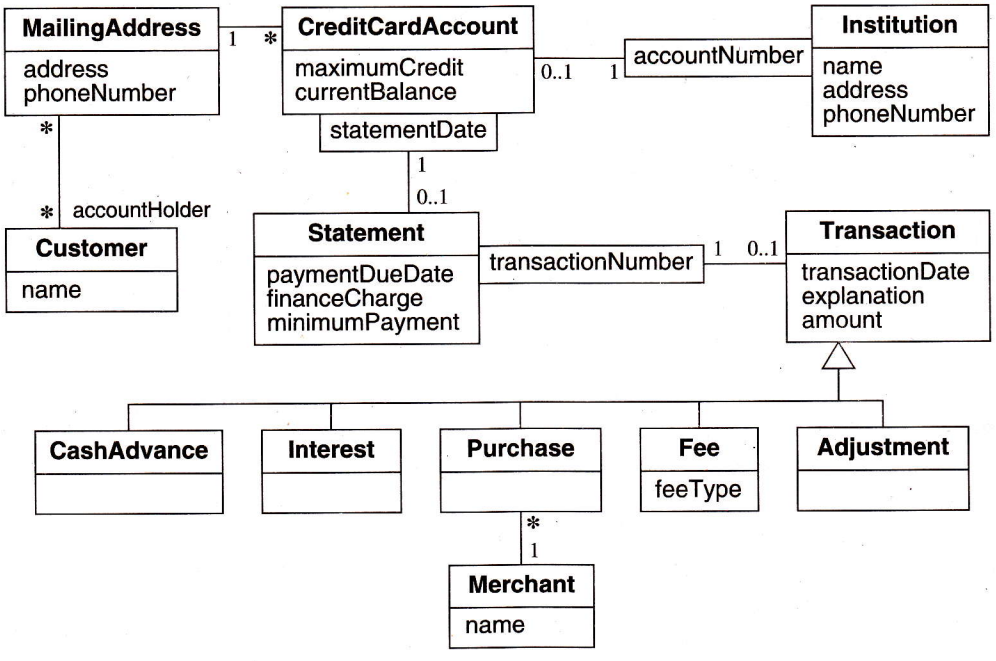


Figure 3.27 Class model for managing credit card accounts

- 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.

3.5.2 Building OCL Expressions

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

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

The example in Figure 3.28 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

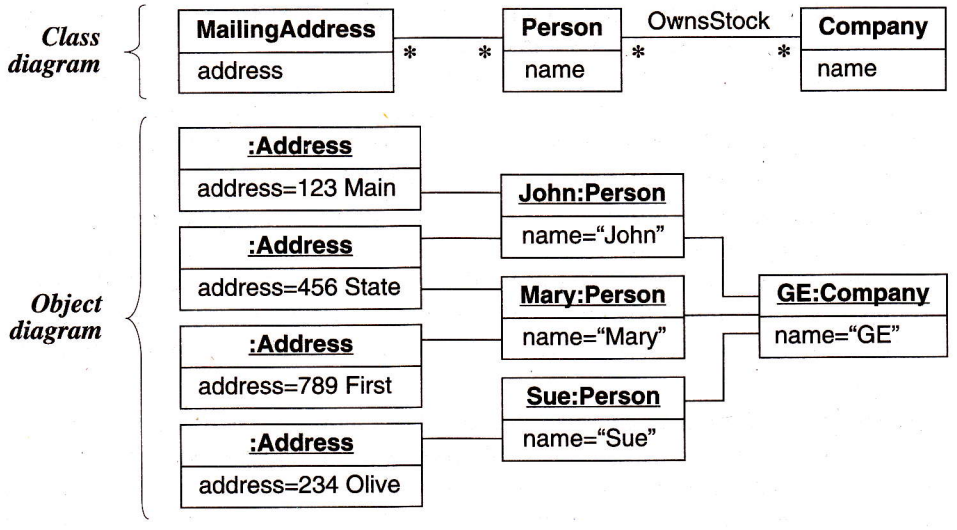


Figure 3.28 A sample model and examples. Traversal of multiple associations can yield a bag.

these three persons and traversing to mailing address, we get a bag obtaining the address *456 State* twice.

[Warmer-99] does not mention null values, since they only discuss the specification of constraints for a correctly implemented system. (***Null*** is a special value denoting that an attribute value is unknown or not applicable.) Handling of exceptions and run-time errors is also outside the scope of their book.

In contrast, the purpose in this chapter is not to specify constraints, but rather to discuss navigation of class models. Nulls do not arise for properly phrased and valid constraints. But they certainly do arise with model navigation. For example, a person may lack a mailing address. We extend the meaning of OCL expressions to accommodate nulls—a traversal may yield a null value, and an OCL expression evaluates to null if the source object is null.

3.5.3 Examples of OCL Expressions

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

- What transactions occurred 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. (Traversal of the two associations results in a set, rather than a bag, because both associations are one-to-many.) Then we use the OCL *select* operator (a collection operator) 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*. (Traversal results in a set, rather than a bag, because all three associations are one-to-many.) The OCL *select* operator finds the transactions within the time interval bounded by *aStartDate* and *anEndDate*. (We choose to make the time interval more general than *last year*.) Then we find the amount for each transaction and compute the total with the OCL *sum* operator (a collection 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 traverses from a *Merchant* object to *Purchase*. The OCL *select* operator finds the transactions within the time interval bounded by *aStartDate* and *anEndDate*. (Traversal across a generalization, from *Purchase* to *Transaction*, is implicit in the OCL.) For these transactions, we then traverse to *Statement*, then to *CreditCardAccount*, then to *MailingAddress*, and finally to *Customer*. The association from *MailingAddress* to *Customer* is many-to-many, so traversal to *Customer* yields a bag. The OCL *asSet* operator converts a bag of customers to a set of customers, resulting in our answer.

- 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 the set of *MailingAddress* objects, we find a set of *CreditCardAccount* objects. (This traversal yields a set, and not a bag, because each *CreditCardAccount* pertains to a single *MailingAddress*.) 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 traverses 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.

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

Keep in mind that the OCL was originally intended as a constraint language (see Chapter 4). However, as we explain here, the OCL is also useful for navigating models.