4

Advanced Class Modeling

Chapter 4 continues our discussion of class modeling concepts with a treatment of advanced topics. This chapter provides subtleties for improved modeling that you can skip upon a first reading of this book.

4.1 Advanced Object and Class Concepts

4.1.1 Enumerations

A data type is a description of values. Data types include numbers, strings, and enumerations. An ***enumeration*** is a data type that has a finite set of values. For example, the attribute *accessPermission* in Figure 3.17 is an enumeration with possible values that include *read* and *read-write*. Figure 3.25 also has some enumerations that Figure 4.1 illustrates. *Figure.penType* is an enumeration that includes *solid*, *dashed*, and *dotted*. *TwoDimensional.fillType* is an enumeration that includes *solid*, *grey*, *none*, *horizontal lines*, and *vertical lines*.

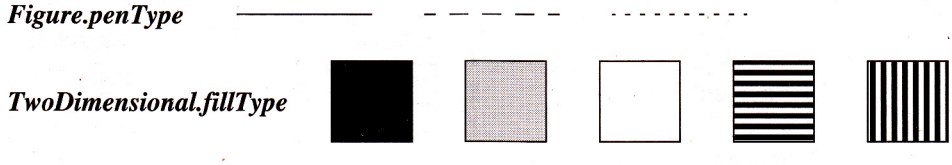

Figure 4.1 Examples of enumerations. Enumerations often occur and are important to users. Implementations must enforce the finite set of values.

When constructing a model, you should carefully note enumerations, because they often occur and are important to users. Enumerations are also significant for an implementation;

you may display the possible values with a pick list and you must restrict data to the legitimate values.

Do not use a generalization to capture the values of an enumerated attribute. An enumeration is merely a list of values; generalization is a means for structuring the description of objects. You should introduce generalization only when at least one subclass has significant attributes, operations, or associations that do not apply to the superclass. As Figure 4.2 shows, you should not introduce a generalization for *Card*, because most games do not differentiate the behavior of spades, clubs, hearts, and diamonds.

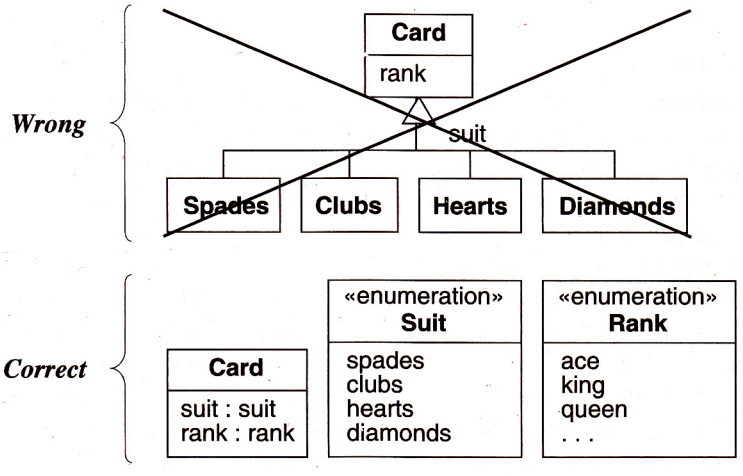

Figure 4.2 Modeling enumerations. Do not use a generalization to capture the values of an enumerated attribute.

In the UML an enumeration is a data type. You can declare an enumeration by listing the keyword *enumeration* in guillemets («») above the enumeration name in the top section of a box. The second section lists the enumeration values.

4.1.2 Multiplicity

Multiplicity is a constraint on the cardinality of a set. Chapter 3 explained multiplicity for associations. Multiplicity also applies to attributes.

It is often helpful to specify multiplicity for an attribute, especially for database applications. ***Multiplicity for an attribute*** specifies the number of possible values for each instantiation of an attribute. The most common specifications are a mandatory single value [1], an optional single value [0..1], and many [*]. Multiplicity specifies whether an attribute is mandatory or optional (in database terminology whether an attribute can be null). Multiplicity also indicates if an attribute is single valued or can be a collection. If not specified, an attribute is assumed to be a mandatory single value ([1]). In Figure 4.3 a person has one name, one or more addresses, zero or more phone numbers, and one birthdate.

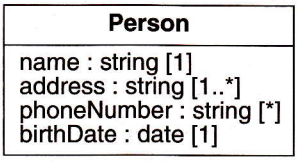

Figure 4.3 Multiplicity for attributes. You can specify whether an attribute is single or multivalued, mandatory or optional.

4.1.3 Scope

Chapter 3 presented features for individual objects. This is the default usage, but there can also be features for an entire class. The ***scope*** indicates if a feature applies to an object or a class. An underline distinguishes features with class scope (static) from those with object scope. Our convention is to list attributes and operations with class scope at the top of the attribute and operation boxes, respectively.

It is acceptable to use an attribute with class scope to hold the ***extent*** of a class (the set of objects for a class)—this is common with OO databases. Otherwise, you should avoid attributes with class scope because they can lead to an inferior model. It is better to model groups explicitly and assign attributes to them. For example, the upper model in Figure 4.4 shows a simple model of phone mail. Each message has an owner mailbox, date recorded, time recorded, priority, message contents, and a flag indicating if it has been received. A message may have a mailbox as the source or it may be from an external call. Each mailbox has a phone number, password, and recorded greeting. For the *PhoneMessage* class we can store the maximum duration for a message and the maximum days a message will be retained. For the *PhoneMailbox* class we can store the maximum number of messages that can be stored.

The upper model is inferior, however, because the maximum duration, maximum days retained, and maximum message count have a single value for the entire phone mail system. In the lower model these limits can vary for different kinds of users, yielding a more flexible and extensible phone mail system.

In contrast to attributes, it is acceptable to define operations of class scope. The most common use of class-scoped operations is to create new instances of a class. Sometimes it is convenient to define class-scoped operations to provide summary data. You should be careful with the use of class-scoped operations for distributed applications.

4.1.4 Visibility

Visibility refers to the ability of a method to reference a feature from another class and has the possible values of *public*, *protected*, *private*, and *package*. The precise meaning depends on the programming language. (See Chapter 18 for details.) Any method can freely access ***public*** features. Only methods of the containing class and its descendants via inheritance can access ***protected*** features. (Protected features also have package accessibility in Java.) Only methods of the containing class can access ***private*** features. Methods of classes defined in the same package as the target class can access ***package*** features.

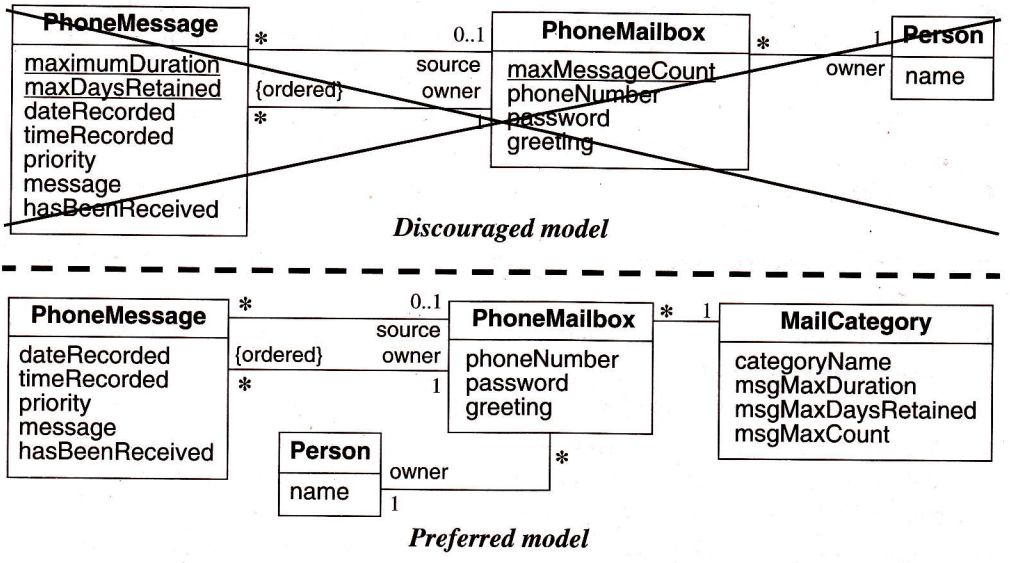

Figure 4.4 Attribute scope. Instead of assigning attributes to classes, model groups explicitly.

The UML denotes visibility with a prefix. The character "+" precedes public features. The character "#" precedes protected features. The character "-" precedes private features. And the character "~" precedes package features. The lack of a prefix reveals no information about visibility.

There are several issues to consider when choosing visibility.

- **Comprehension.** You must understand all public features to understand the capabilities of a class. In contrast, you can ignore private, protected, and package features—they are merely an implementation convenience.
- **Extensibility.** Many classes can depend on public methods, so it can be highly disruptive to change their signature (number of arguments, types of arguments, type of return value). Since fewer classes depend on private, protected, and package methods, there is more latitude to change them.
- **Context.** Private, protected, and package methods may rely on preconditions or state information created by other methods in the class. Applied out of context, a private method may calculate incorrect results or cause the object to fail.

4.2 Association Ends

As the name implies, an ***association end*** is an end of an association. A binary association has two ends, a ternary association (Section 4.3) has three ends, and so forth. Chapter 3 discussed the following properties.

- **Association end name**. An association end may have a name. The names disambiguate multiple references to a class and facilitate navigation. Meaningful names often arise, and it is useful to place the names within the proper context.
- **Multiplicity**. You can specify multiplicity for each association end. The most common multiplicities are "1" (exactly one), "0..1" (at most one), and "*" ("many"—zero or more).
- **Ordering**. The objects for a "many" association end are usually just a set. However, sometimes the objects have an explicit order.
- **Bags and sequences**. The objects for a "many" association end can also be a bag or sequence.
- **Qualification**. One or more qualifier attributes can disambiguate the objects for a "many" association end.

Association ends have some additional properties.

- **Aggregation**. The association end may be an aggregate or constituent part (Section 4.4). Only a binary association can be an aggregation; one association end must be an aggregate and the other must be a constituent.
- **Changeability**. This property specifies the update status of an association end. The possibilities are *changeable* (can be updated) and *readonly* (can only be initialized).
- **Navigability**. Conceptually, an association may be traversed in either direction. However, an implementation may support only one direction. The UML shows navigability with an arrowhead on the association end attached to the target class. Arrowheads may be attached to zero, one, or both ends of an association.
- **Visibility**. Similar to attributes and operations (Section 4.1.4), association ends may be *public*, *protected*, *private*, or *package*.

4.3 N-ary Associations

Chapter 3 presented binary associations (associations between two classes). However, you may occasionally encounter ***n-ary associations*** (associations among three or more classes.) You should try to avoid n-ary associations—most of them can be decomposed into binary associations, with possible qualifiers and attributes. Figure 4.5 shows an association that at first glance might seem to be an n-ary but can readily be restated as binary associations.

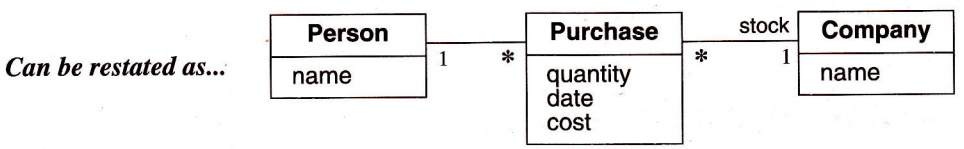

Figure 4.5 Restating an n-ary association. You can decompose most n-ary associations into binary associations.

Figure 4.6 shows a genuine n-ary (ternary) association: Programmers use computer languages on projects. This n-ary association is an atomic unit and cannot be subdivided into binary associations without losing information. A programmer may know a language and work on a project, but might not use the language on the project. The UML symbol for n-ary associations is a diamond with lines connecting to related classes. If the association has a name, it is written in italics next to the diamond.

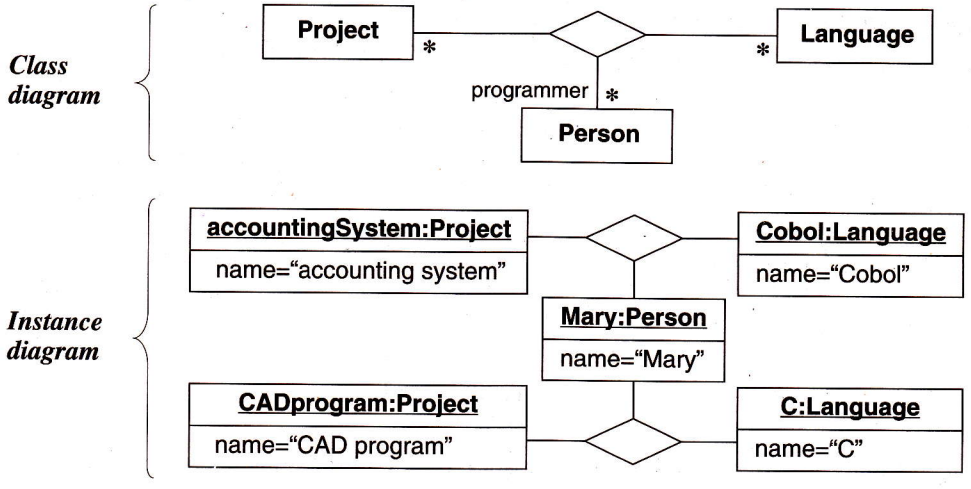

Figure 4.6 Ternary association and links. An n-ary association can have association end names, just like a binary association.

As Figure 4.6 illustrates, an n-ary association can have a name for each end just like a binary association. End names are necessary if a class participates in an n-ary association more than once. You cannot traverse n-ary associations from one end to another as with binary associations, so end names do not represent pseudo attributes of the participating classes. The OCL [Warmer-99] does not define notation for traversing n-ary associations.

Figure 4.7 shows another ternary association: A professor teaches a listed course during a semester. The resulting delivered course has a room number and any number of textbooks.

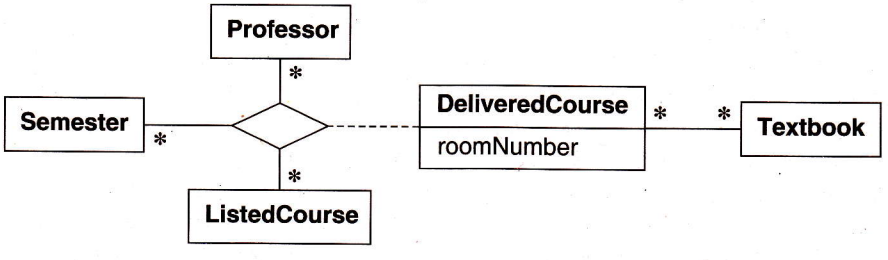

Figure 4.7 Another ternary association. N-ary associations are full-fledged associations and can have association classes.

The typical programming language cannot express n-ary associations. Thus if you are programming, you will need to promote n-ary associations to classes as Figure 4.8 does for *DeliveredClass*. Be aware that you change the meaning of a model, when you promote an n-ary association to a class. An n-ary association enforces that there is at most one link for each combination—for each combination of *Professor*, *Semester*, and *ListedCourse* in Figure 4.7 there is one *DeliveredCourse*. In contrast a promoted class permits any number of links—for each combination of *Professor*, *Semester*, and *ListedCourse* in Figure 4.8 there can be many *DeliveredCourses*. If you were implementing Figure 4.8, special application code would have to enforce the uniqueness of *Professor* + *Semester* + *ListedCourse*.

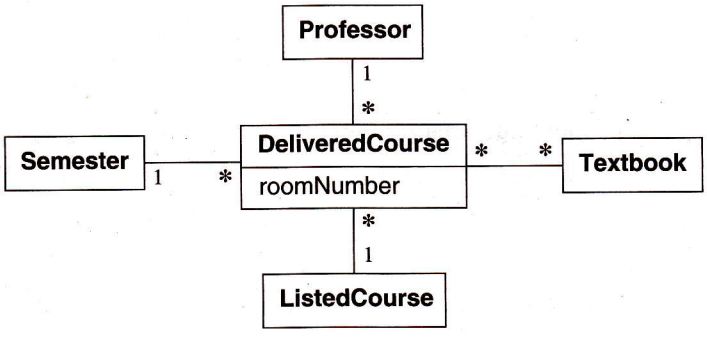

Figure 4.8 Promoting an n-ary association. Programming languages cannot express n-ary associations, so you must promote them to classes.

4.4 Aggregation

Aggregation is a strong form of association in which an aggregate object is *made of* constituent parts. Constituents are *part of* the aggregate. The aggregate is semantically an extended object that is treated as a unit in many operations, although physically it is made of several lesser objects.

We define an aggregation as relating an assembly class to *one* constituent part class. An assembly with many kinds of constituent parts corresponds to many aggregations. For example, a *LawnMower* consists of a *Blade*, an *Engine*, many *Wheels*, and a *Deck*. *LawnMower* is the assembly and the other parts are constituents. *LawnMower* to *Blade* is one aggregation, *LawnMower* to *Engine* is another aggregation, and so on. We define each individual pairing as an aggregation so that we can specify the multiplicity of each constituent part within the assembly. This definition emphasizes that aggregation is a special form of binary association.

The most significant property of aggregation is ***transitivity***—that is, if *A* is part of *B* and *B* is part of *C*, then *A* is part of *C*. Aggregation is also ***antisymmetric***—that is, if *A* is part of *B*, then *B* is not part of *A*. Many aggregate operations imply transitive closure* and operate on both direct and indirect parts.

4.4.1 Aggregation Versus Association

Aggregation is a special form of association, not an independent concept. Aggregation adds semantic connotations. If two objects are tightly bound by a part-whole relationship, it is an aggregation. If the two objects are usually considered as independent, even though they may often be linked, it is an association. Some tests include:

- Would you use the phrase *part of*?
- Do some operations on the whole automatically apply to its parts?
- Do some attribute values propagate from the whole to all or some parts?
- Is there an intrinsic asymmetry to the association, where one class is subordinate to the other?

Aggregations include bill-of-materials, part explosions, and expansions of an object into constituent parts. Aggregation is drawn like association, except a small diamond indicates the assembly end. In Figure 4.9 a lawn mower consists of one blade, one engine, many wheels, and one deck. The manufacturing process is flexible and largely combines standard parts, so blades, engines, wheels, and decks pertain to multiple lawn mower designs.

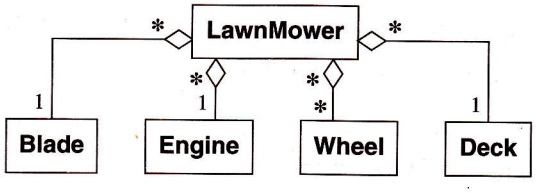

Figure 4.9 Aggregation. Aggregation is a kind of association in which an aggregate object is made of constituent parts.

The decision to use aggregation is a matter of judgment and can be arbitrary. Often it is not obvious if an association should be modeled as an aggregation. To a large extent this kind of uncertainty is typical of modeling; modeling requires seasoned judgment and there are few hard and fast rules. Our experience has been that if you exercise careful judgment and are consistent, the imprecise distinction between aggregation and ordinary association does not cause problems in practice.

4.4.2 Aggregation Versus Composition

The UML has two forms of part-whole relationships: a general form called *aggregation* and a more restrictive form called *composition*.

---

* Transitive closure is a term from graph theory. If E denotes an edge and N denotes a node and S is the set of all pairs of nodes connected by an edge, then S^+ (the transitive closure of S) is the set of all pairs of nodes directly or indirectly connected by a sequence of edges. Thus S^+ includes all nodes that are directly connected, nodes connected by two edges, nodes connected by three edges, and so forth.

Composition is a form of aggregation with two additional constraints. A constituent part can belong to at most one assembly. Furthermore, once a constituent part has been assigned an assembly, it has a coincident lifetime with the assembly. Thus composition implies ownership of the parts by the whole. This can be convenient for programming: Deletion of an assembly object triggers deletion of all constituent objects via composition. The notation for composition is a small *solid* diamond next to the assembly class (vs. a small *hollow* diamond for the general form of aggregation).

In Figure 4.10 a company consists of divisions, which in turn consist of departments; a company is indirectly a composition of departments. A company is not a composition of its employees, since company and person are independent objects of equal stature.

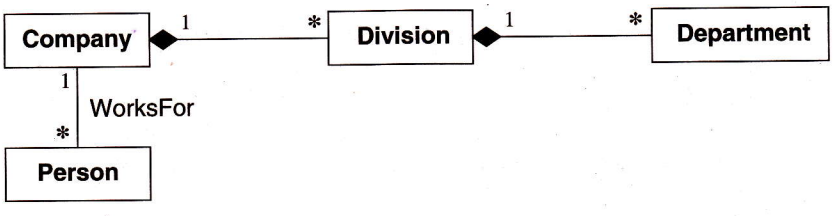

Figure 4.10 Composition. With composition a constituent part belongs to at most one assembly and has a coincident lifetime with the assembly.

4.4.3 *Propagation of Operations*

Propagation (also called ***triggering***) is the automatic application of an operation to a network of objects when the operation is applied to some starting object [Rumbaugh-88].† For example, moving an aggregate moves its parts; the move operation propagates to the parts. Propagation of operations to parts is often a good indicator of aggregation.

Figure 4.11 shows an example of propagation. A person owns multiple documents. Each document consists of paragraphs that, in turn, consist of characters. The copy operation propagates from documents to paragraphs to characters. Copying a paragraph copies all the characters in it. The operation does not propagate in the reverse direction; a paragraph can be copied without copying the whole document. Similarly, copying a document copies the owner link but does not spawn a copy of the person who is owner.

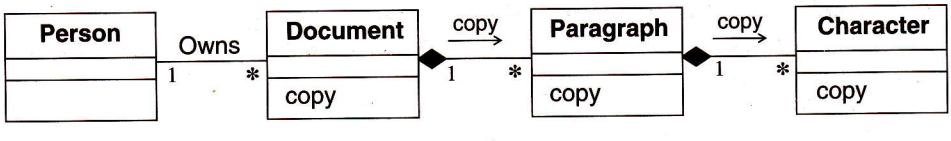

Figure 4.11 Propagation. You can propagate operations across aggregations and compositions.

† The term *association* as used in this book is synonymous with the term *relation* used in [Rumbaugh-88].

Most other approaches present an all-or-nothing option: copy an entire network with deep copy, or copy the starting object and none of the related objects with shallow copy. The concept of propagation of operations provides a concise and powerful way for specifying a continuum of behavior. You can think of an operation as starting at some initial object and flowing from object to object through links according to propagation rules. Propagation is possible for other operations including save/restore, destroy, print, lock, and display.

You can indicate propagation on class models with a small arrow indicating the direction and operation name next to the affected association. The notation binds propagation behavior to an association (or aggregation), direction, and operation. Note that this notation is not part of the UML and is a special notation.

4.5 Abstract Classes

An ***abstract class*** is a class that has no direct instances but whose descendant classes have direct instances. A ***concrete class*** is a class that is instantiable; that is, it can have direct instances. A concrete class may have abstract subclasses (but they, in turn, must have concrete descendants). Only concrete classes may be leaf classes in an inheritance tree.

All the occupations shown in Figure 4.12 are concrete classes. *Butcher, Baker,* and *CandlestickMaker* are concrete classes because they have direct instances. *Worker* also is a concrete class because some occupations may not be specified.

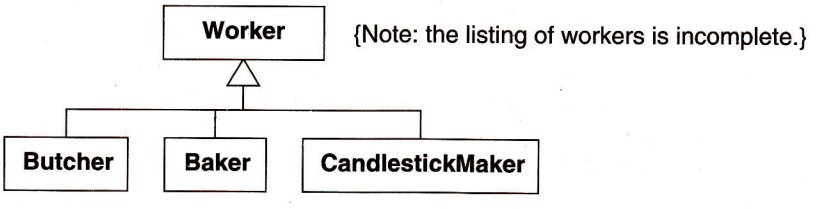

Figure 4.12 Concrete classes. A concrete class is instantiable; that is, it can have direct instances.

Class *Employee* in Figure 4.13 is an example of an abstract class. All employees must be either full-time or part-time. *FullTimeEmployee* and *PartTimeEmployee* are concrete classes because they can be directly instantiated. In the UML notation an abstract class name is listed in an italic font. Or you may place the keyword *{abstract}* below or after the name.

You can use abstract classes to define methods that can be inherited by subclasses. Alternatively, an abstract class can define the signature for an operation without supplying a corresponding method. We call this an ***abstract operation***. (Recall that an operation specifies the form of a function or procedure; a method is the actual implementation.) An abstract operation defines the signature of an operation for which each concrete subclass must provide its own implementation. A concrete class may not contain abstract operations, because objects of the concrete class would have undefined operations.

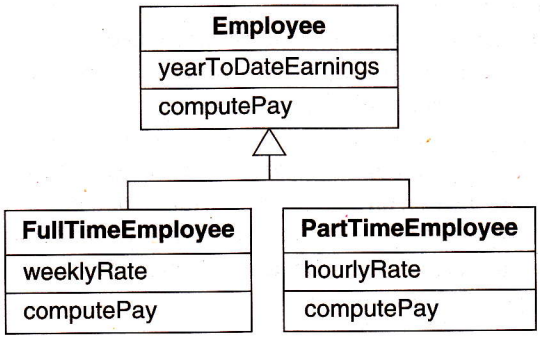

Figure 4.13 Abstract class and abstract operation. An abstract class is a class that has no direct instances

Figure 4.13 shows an abstract operation. An abstract operation is designated by italics or the keyword *{abstract}*. *ComputePay* is an abstract operation of class *Employee*; its signature but not its implementation is defined. Each subclass must supply a method for this operation.

Note that the abstract nature of a class is always provisional, depending on the point of view. You can always refine a concrete class into subclasses, making it abstract. Conversely, an abstract class may become concrete in an application in which the difference among its subclasses is unimportant.

As a matter of style, it is a good idea to avoid concrete superclasses. Then, abstract and concrete classes are readily apparent at a glance; all superclasses are abstract and all leaf subclasses are concrete. Furthermore, you will avoid awkward situations where a concrete superclass must both specify the signature of an operation for descendant classes and also provide an implementation for its concrete instances. You can always eliminate concrete superclasses by introducing an *Other* subclass, as Figure 4.14 shows.

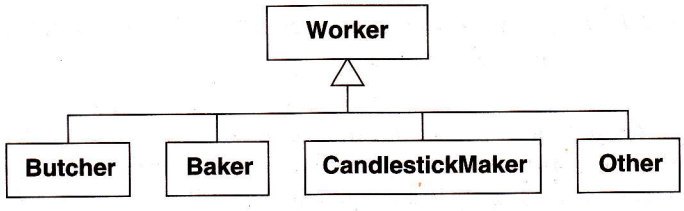

Figure 4.14 Avoiding concrete superclasses. You can always eliminate concrete superclasses by introducing an *Other* subclass.

4.6 Multiple Inheritance

Multiple inheritance permits a class to have more than one superclass and to inherit features from all parents. Then you can mix information from two or more sources. This is a more

complicated form of generalization than single inheritance, which restricts the class hierarchy to a tree. The advantage of multiple inheritance is greater power in specifying classes and an increased opportunity for reuse. The disadvantage is a loss of conceptual and implementation simplicity.

The term *multiple inheritance* is used somewhat imprecisely to mean either the conceptual relationship between classes or the language mechanism that implements that relationship. Whenever possible, we try to distinguish between *generalization* (the conceptual relationship) and *inheritance* (the language mechanism), but the term "multiple inheritance" is more widely used than the term "multiple generalization."

4.6.1 Kinds of Multiple Inheritance

The most common form of multiple inheritance is from sets of disjoint classes. Each subclass inherits from one class in each set. In Figure 4.15 *FullTimeIndividualContributor* is both *FullTimeEmployee* and *IndividualContributor* and combines their features. *FullTimeEmployee* and *PartTimeEmployee* are disjoint; each employee must belong to exactly one of these. Similarly, *Manager* and *IndividualContributor* are also disjoint and each employee must be one or the other. The model does not show it, but we could define three additional combinations: *FullTimeManager*, *PartTimeIndividualContributor*, and *PartTimeManager*. The appropriate combinations depend on the needs of an application.

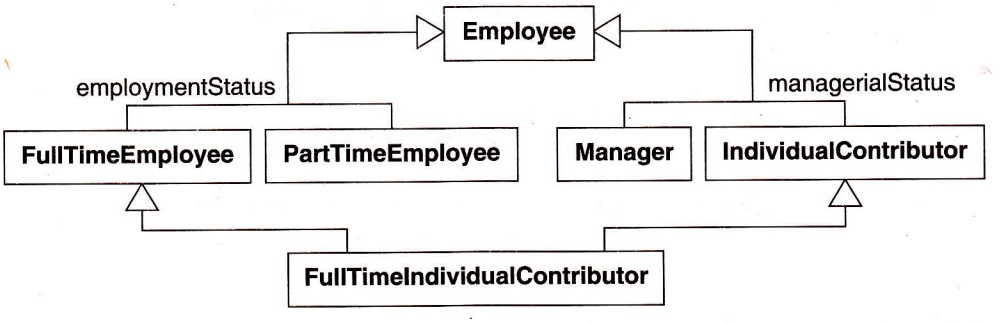

Figure 4.15 Multiple inheritance from disjoint classes. This is the most common form of multiple inheritance.

Each generalization should cover a single aspect. You should use multiple generalizations if a class can be refined on several distinct and independent aspects. In Figure 4.15, class *Employee* independently specializes on employment status and managerial status. Consequently the model has two separate generalization sets.

A subclass inherits a feature from the same ancestor class found along more than one path only once; it is the same feature. For example, in Figure 4.15 *FullTimeIndividualContributor* inherits *Employee* features along two paths, via *employmentStatus* and *managerialStatus*. However, each *FullTimeIndividualContributor* has only a single copy of *Employee* features.

Conflicts among parallel definitions create ambiguities that implementations must resolve. In practice, you should avoid such conflicts in models or explicitly resolve them, even if a particular language provides a priority rule for resolving conflicts. For example, suppose that *FullTimeEmployee* and *IndividualContributor* both have an attribute called *name*. *FullTimeEmployee.name* could refer to the person's full name while *IndividualContributor.name* might refer to the person's title. In principle, there is no obvious way to resolve such clashes. The best solution is to try to avoid them by restating the attributes as *FullTimeEmployee.personName* and *IndividualContributor.title*.

Multiple inheritance can also occur with overlapping classes. In Figure 4.16, *AmphibiousVehicle* is both *LandVehicle* and *WaterVehicle*. *LandVehicle* and *WaterVehicle* overlap, because some vehicles travel on both land and water. The UML uses a constraint (see Section 4.9) to indicate an overlapping generalization set; the notation is a dotted line cutting across the affected generalizations with keywords in braces. In this example, *overlapping* means that an individual vehicle may belong to more than one of the subclasses. *Incomplete* means that all possible subclasses of vehicle have not been explicitly named.

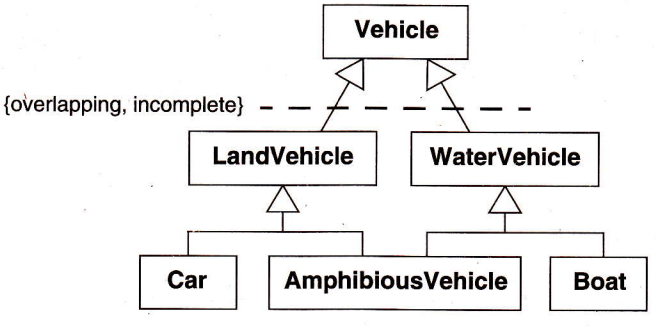

Figure 4.16 Multiple inheritance from overlapping classes. This form of multiple inheritance occurs less often than with disjoint classes.

4.6.2 Multiple Classification

An instance of a class is inherently an instance of all ancestors of the class. For example, an instructor could be both faculty and student. But what about a Harvard Professor taking classes at MIT? There is no class to describe the combination (it would be artificial to make one). This is an example of multiple classification, in which one instance happens to participate in two overlapping classes.

The UML permits multiple classification, but most OO languages handle it poorly. As Figure 4.17 shows, the best approach using conventional languages is to treat *Person* as an object composed of multiple *UniversityMember* objects. This workaround replaces inheritance with delegation (discussed in the next section). This is not totally satisfactory, because there is a loss of identity between the separate roles, but the alternatives involve radical changes in many programming languages [McAllester-86].

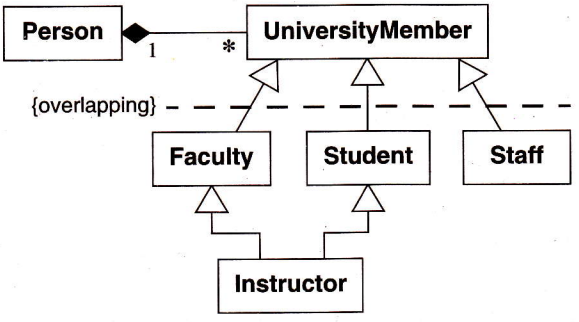

Figure 4.17 Workaround for multiple classification. OO languages do not handle this well, so you must use a workaround.

4.6.3 Workarounds

Dealing with lack of multiple inheritance is really an implementation issue, but early restructuring of a model is often the easiest way to work around its absence. We list some restructuring techniques below. Two of the approaches make use of ***delegation***, which is an implementation mechanism by which an object forwards an operation to another object for execution. See Chapter 15 for a further discussion of delegation.

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

  For example, in Figure 4.18 *EmployeeEmployment* becomes a superclass of *FullTimeEmployee* and *PartTimeEmployee*. *EmployeeManagement* becomes a superclass of *Manager* and *IndividualContributor*. Then you can model *Employee* as a composition of *EmployeeEmployment* and *EmployeeManagement*. An operation sent to an *Employee* object would have to be redirected to the *EmployeeEmployment* or *EmployeeManagement* part by the *Employee* class.

  In this approach, you need not create the various combinations (such as *FullTimeIndividualContributor*) as explicit classes. All combinations of subclasses from the different generalizations are possible.

- **Inherit the most important class and delegate the rest.** Figure 4.19 preserves identity and inheritance across the most important generalization. You degrade the remaining generalizations to composition and delegate their operations as in the previous alternative.

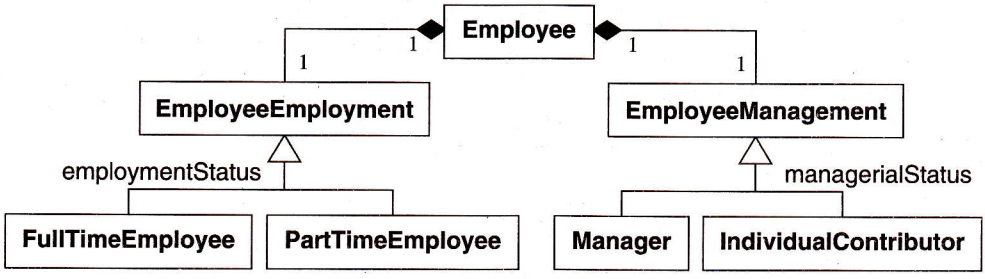

Figure 4.18 Workaround for multiple inheritance—delegation

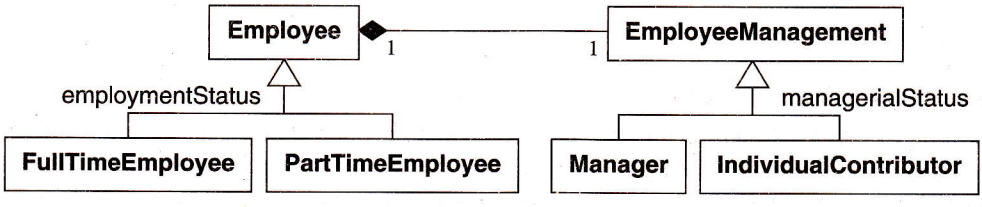

Figure 4.19 Workaround for multiple inheritance—inheritance and delegation

- **Nested generalization.** Factor on one generalization first, then the other. This approach multiplies out all possible combinations. For example, in Figure 4.20 under *FullTimeEmployee* and *PartTimeEmployee*, add two subclasses for managers and individual contributors. This preserves inheritance but duplicates declarations and code and violates the spirit of OO programming.

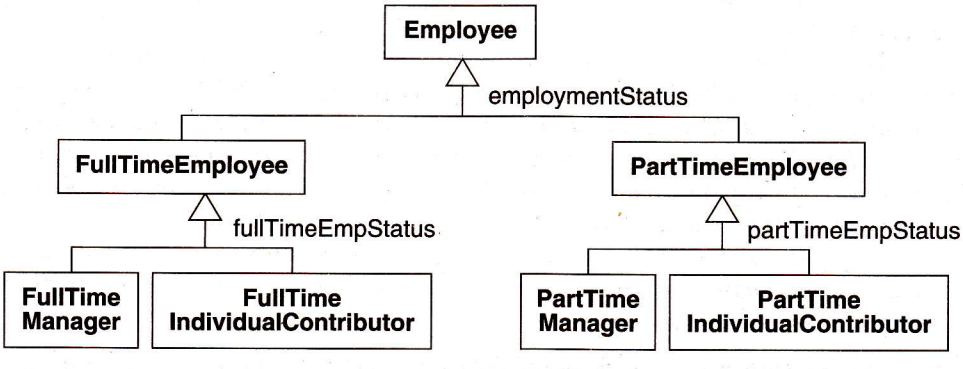

Figure 4.20 Workaround for multiple inheritance—nested generalization

Any of these workarounds can be made to work, but they all compromise logical structure and maintainability. There are several issues to consider when selecting the best workaround.

- **Superclasses of equal importance**. If a subclass has several superclasses, all of equal importance, it may be best to use delegation (Figure 4.18) and preserve symmetry in the model.
- **Dominant superclass**. If one superclass clearly dominates and the others are less important, preserve inheritance through this path (Figure 4.19 or Figure 4.20).
- **Few subclasses**. If the number of combinations is small, consider nested generalization (Figure 4.20). If the number of combinations is large, avoid it.
- **Sequencing generalization sets**. If you use nested generalization (Figure 4.20), factor on the most important criterion first, the next most important second, and so forth.
- **Large quantities of code**. Try to avoid nested generalization (Figure 4.20) if you must duplicate large quantities of code.
- **Identity**. Consider the importance of maintaining strict identity. Only nested generalization (Figure 4.20) preserves this.

4.7 Metadata

Metadata is data that describes other data. For example, a class definition is metadata. Models are inherently metadata, since they describe the things being modeled (rather than *being* the things). Many real-world applications have metadata, such as parts catalogs, blueprints, and dictionaries. Computer-language implementations also use metadata heavily.

Figure 4.21 shows an example of metadata and data. A car model has a model name, year, base price, and a manufacturer. Some examples of car models are a 1969 Ford Mustang and a 1975 Volkswagen Rabbit. A physical car has a serial number, color, options, and an owner. As an example of physical cars, John Doe may own a blue Ford with serial number *1FABP* and a red Volkswagen with serial number *7E81F*. A car model describes many physical cars and holds common data. A car model is metadata relative to a physical car, which is data.

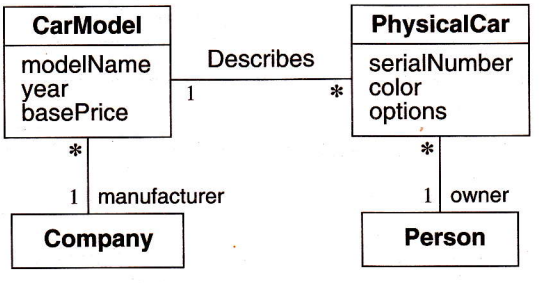

Figure 4.21 Example of metadata. Metadata often arises in applications.

You can also consider classes as objects, but classes are meta-objects and not real-world objects. Class descriptor objects have features, and they in turn have their own classes, which

are called ***metaclasses***. Treating everything as an object provides a more uniform implementation and greater functionality for solving complex problems. Languages vary in their accessibility for metadata. Some languages, like Lisp and Smalltalk, let metadata be inspected and altered by programs at run time. In contrast, languages like C++ and Java deal with metadata at compile time but do not make the metadata explicitly available at run time.

4.8 Reification

Reification is the promotion of something that is not an object into an object. Reification is a helpful technique for meta applications because it lets you shift the level of abstraction. On occasion it is useful to promote attributes, methods, constraints, and control information into objects so you can describe and manipulate them as data.

As an example of reification, consider a database manager. A developer could write code for each application so that it can read and write from files. Instead, for many applications, it is a better idea to reify the notion of data services and use a database manager. A database manager has abstract functionality that provides a general-purpose solution to accessing data reliably and quickly for multiple users.

For another example, consider state-transition diagrams (see the next two chapters). You can use a state-transition diagram to specify control and then implement it by writing the corresponding code. Alternatively, you can prepare a metamodel and store a state-transition model as data. A general-purpose interpreter reads the contents of the metamodel and executes the intent.

Figure 4.22 promotes the *substanceName* attribute to a class to capture the many-to-many relationship between *Substance* and *SubstanceName*. A chemical substance may have multiple aliases. For example, propylene may be referred to as *propylene* and C_3H_6. Also, an alias may pertain to multiple chemical substances. Various mixtures of ethylene glycol and automotive additives may have the alias of *antifreeze*.

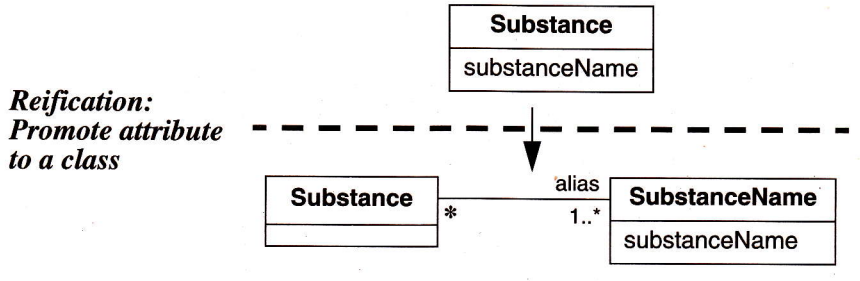

Figure 4.22 Reification. Reification is the promotion of something that is not an object into an object and can be helpful for meta applications.

4.9 Constraints

A ***constraint*** is a boolean condition involving model elements, such as objects, classes, attributes, links, associations, and generalization sets. A constraint restricts the values that elements can assume. You can express constraints with natural language or a formal language such as the Object Constraint Language (OCL) [Warmer-99].

4.9.1 Constraints on Objects

Figure 4.23 shows several examples of constraints. No employee's salary can exceed the salary of the employee's boss (a constraint between two things at the same time). No window can have an aspect ratio (length/width) of less than 0.8 or greater than 1.5 (a constraint between attributes of a single object). The priority of a job may not increase (constraint on the same object over time). You may place simple constraints in class models.

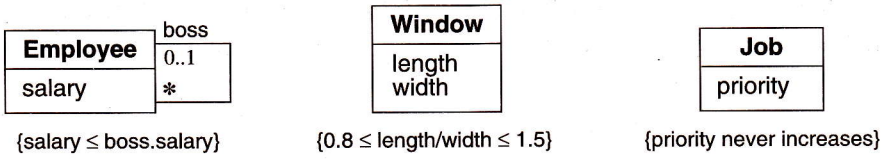

Figure 4.23 Constraints on objects. The structure of a model expresses many constraints, but sometimes it is helpful to add explicit constraints.

4.9.2 Constraints on Generalization Sets

Class models capture many constraints through their very structure. For example, the semantics of generalization imply certain structural constraints. With single inheritance the subclasses are mutually exclusive. Furthermore, each instance of an abstract superclass corresponds to exactly one subclass instance. Each instance of a concrete superclass corresponds to at most one subclass instance.

Figure 4.16 and Figure 4.17 use a constraint to help express multiple inheritance. The UML defines the following keywords for generalization sets.

- **Disjoint**. The subclasses are mutually exclusive. Each object belongs to exactly one of the subclasses.
- **Overlapping**. The subclasses can share some objects. An object may belong to more than one subclass.
- **Complete**. The generalization lists all the possible subclasses.
- **Incomplete**. The generalization may be missing some subclasses.

4.9.3 Constraints on Links

Multiplicity is a constraint on the cardinality of a set. Multiplicity for an association restricts the number of objects related to a given object. Multiplicity for an attribute specifies the number of values that are possible for each instantiation of an attribute.

Qualification also constrains an association. A qualifier attribute does not merely describe the links of an association but is also significant in resolving the "many" objects at an association end.

An association class implies a constraint. An association class is a class in every right; for example, it can have attributes and operations, participate in associations, and participate in generalizations. But an association class has a constraint that an ordinary class does not; it derives identity from instances of the related classes.

An ordinary association presumes no particular order on the objects of a "many" end. The constraint *{ordered}* indicates that the elements of a "many" association end have an explicit order that must be preserved.

Figure 4.24 shows an explicit constraint that is not part of the model's structure. The chair of a committee must be a member of the committee; the *ChairOf* association is a subset of the *MemberOf* association.

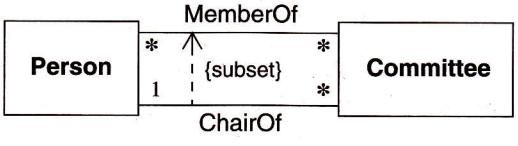

Figure 4.24 Subset constraint between associations.

4.9.4 Use of Constraints

We favor expressing constraints in a declarative manner. Declaration lets you express a constraint's intent, without supposing an implementation. Typically, you will need to convert constraints to procedural form before you can implement them in a programming language, but this conversion is usually straightforward.

Constraints provide one criterion for measuring the quality of a class model; a "good" class model captures many constraints through its structure. It often requires several iterations to get the structure of a model right from the perspective of constraints. Also, in practice, you cannot enforce every constraint with a model's structure, but you should try to enforce the important ones.

The UML has two alternative notations for constraints. You can either delimit a constraint with braces or place it in a "dog-eared" comment box (Figure 4.26). Either way, you should try to position constraints near the affected elements. You can use dashed lines to connect constrained elements. A dashed arrow can connect a constrained element to the element on which it depends.

4.10 Derived Data

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

Figure 4.25 shows a derived attribute. Age can be derived from birthdate and the current date.

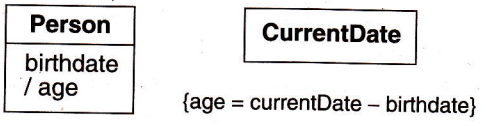

Figure 4.25 Derived attribute. A derived attribute is a function of one or more elements.

In Figure 4.26, a machine consists of several assemblies that in turn consist of parts. An assembly has a geometrical offset with respect to machine coordinates; each part has an offset with respect to assembly coordinates. We can define a coordinate system for each part that is derived from machine coordinates, assembly offset, and part offset. This coordinate system can be represented as a derived class called *Offset* related to each part by a derived association called *NetOffset*.

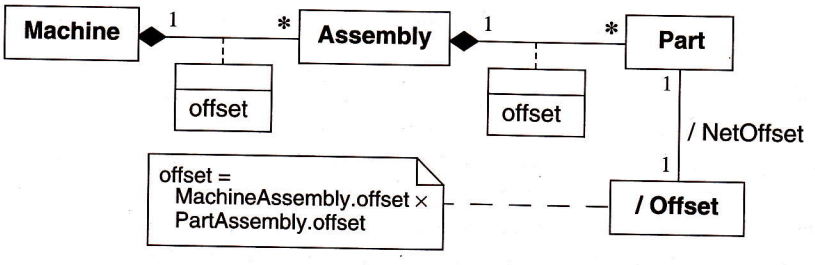

Figure 4.26 Derived object and association. Derived data can complicate implementation, so only use derived data where it truly is compelling.

It is useful to distinguish operations with side effects from those that merely compute a functional value without modifying any objects. The latter kind of operation is called a ***query***. You can regard queries with no arguments except the target object as derived attributes. For example, you can compute the width of a box from the positions of its sides. In many cases, an object has a set of attributes with interrelated values, of which only a fixed number of values can be chosen independently. A class model should generally distinguish independent *base attributes* from dependent *derived attributes*. The choice of base attributes is arbitrary but should be made to avoid overspecifying the state of the object.

Some developers tend to include many derived elements. Generally, this is not helpful and clutters a model. You should only include derived elements when they are important application concepts or substantially ease implementation. It can be quite difficult to keep derived elements consistent with the base data, so only use derived elements for implementation where they are clearly compelling.

4.11 Packages

You can fit a class model on a single page for many small and medium-sized problems. However, it is often difficult to grasp the entirety of a large model. We recommend that you partition large models so that people can understand them.

A ***package*** is a group of elements (classes, associations, generalizations, and lesser packages) with a common theme. A package partitions a model, making it easier to understand and manage. Large applications may require several tiers of packages. Packages form a tree with increasing abstraction toward the root, which is the application, the top-level package. As Figure 4.27 shows, the notation for a package is a box with a tab. The purpose of the tab is to suggest the enclosed contents, like a tabbed folder.

Figure 4.27 Notation for a package. Packages let you organize large models so that persons can more readily understand them.

There are various themes for forming packages: dominant classes, dominant relationships, major aspects of functionality, and symmetry. For example, many business systems have a *Customer* package or a *Part* package; *Customer* and *Part* are dominant classes that are important to the business of a corporation and appear in many applications. In an engineering application we used a dominant relationship, a large generalization for many kinds of equipment, to divide a class model into packages. Equipment was the focus of the model, and the attributes and relationships varied greatly across types of equipment. You could divide the class model of a compiler into packages for lexical analysis, parsing, semantic analysis, code generation, and optimization. Once some packages have been established, symmetry may suggest additional packages.

We can offer the following tips for devising packages.

- **Carefully delineate each package's scope.** The precise boundaries of a package are a matter of judgment. Like other aspects of modeling, defining the scope of a package requires planning and organization. Make sure that class and association names are unique within each package, and use consistent names across packages as much as possible.
- **Define each class in a single package.** The defining package should show the class name, attributes, and operations. Other packages that refer to a class can use a class icon,

a box that contains only the class name. This convention makes it easier to read class models, because a class is prominent in its defining package. Readers are not distracted by definitions that may be inconsistent or misled by forgetting a prior class definition. This convention also makes it easier to develop packages concurrently.

- **Make packages cohesive.** Associations and generalizations should normally appear in a single package, but classes can appear in multiple packages, helping to bind them. Try to limit appearances of classes in multiple packages. Typically no more than 20–30% of classes should appear in multiple packages.

4.12 Practical Tips

Here are tips for constructing class models in addition to those from Chapter 3.

- **Enumerations.** When constructing a model, you should declare enumerations and their values, because they often occur and are important to users. Do not create unnecessary generalizations for attributes that are enumerations. Only specialize a class when the subclasses have distinct attributes, operations, or associations. (Section 4.1.1)
- **Class-scoped (static) attributes.** It is acceptable to use an attribute with class scope to hold the extent of a class. Otherwise, you should avoid attributes with class scope because they can lead to an inferior model. You can improve a model by explicitly modeling groups and assigning attributes to them. (Section 4.1.3)
- **N-ary associations.** Try to avoid n-ary associations. Most n-ary associations can be decomposed into binary associations. (Section 4.3)
- **Concrete superclasses.** As a matter of style, it is best to avoid concrete superclasses. Then, abstract and concrete classes are readily apparent at a glance—all superclasses are abstract and all leaf subclasses are concrete. You can always eliminate concrete superclasses by introducing an *Other* subclass. (Section 4.5)
- **Multiple inheritance.** Limit your use of multiple inheritance to that which is essential for a model. (Section 4.6)
- **Constraints.** You may be able to restructure a class model to improve clarity and capture additional constraints. (Section 4.9)
- **Derived elements.** You should always indicate when an element is derived. Use derived elements sparingly. (Section 4.10)
- **Large models.** Use packages to organize large models so that the reader can understand portions of the model at a time, rather than having to deal with the whole model at once. (Section 4.11)
- **Defining classes.** Define each class in a single package and show its features there. Other packages that refer to the class should use a class icon, a box that contains only the class name. This convention makes it easier to read class models and facilitates concurrent development. (Section 4.11)

4.13 Chapter Summary

This chapter covers several diverse topics that explain subtleties of class modeling. You will not need these concepts for simple models, but they can be important for complex applications. Remember, application needs should drive the content of any model. Only use the advanced concepts in this chapter if they truly add to your application, either by improving clarity, tightening structural constraints, or permitting expression of a difficult concept.

A data type is a description of values; you must assign every attribute a data type before a model can be implemented. Enumerations are a special data type that constrains the permissible values; enumerated values are often prominent in user interfaces.

Multiplicity is a constraint on the cardinality of a set. It applies to attributes as well as associations. Multiplicity for an association restricts the number of objects related to a given object. Multiplicity for an attribute specifies the number of values that are possible for each attribute instantiation.

You should try to avoid n-ary associations—you can decompose most of them into binary associations. Only use n-ary associations that are atomic and cannot be decomposed. Be aware that most programming languages will force you to promote n-ary associations to classes.

Aggregation is a strong form of association in which an aggregate object is made of constituent parts. Aggregation has the properties of transitivity and antisymmetry that differentiate it from association. Operations on an aggregate often propagate to the constituent parts.

Composition is a form of aggregation with two additional constraints. A constituent part can belong to at most one assembly. Furthermore, once a constituent part has been assigned an assembly, it has a coincident lifetime with the assembly. Composition implies ownership of a part by an assembly.

An abstract class has no direct instances. A concrete class may have direct instances. Abstract classes can define methods in one place for use by several subclasses. You can also use abstract classes to define the signature of an operation, leaving the implementation to each subclass.

Multiple inheritance permits a subclass to inherit features from more than one superclass. Each generalization should discriminate a single aspect You should arrange subclasses into more than one generalization if their superclass specializes on more than one aspect. A subclass may combine classes from different generalizations, or it may combine classes from an overlapping generalization, but it may not combine classes from the same disjoint generalization.

Metadata is data that describes other data. Classes are metadata, since they describe objects. Metadata is a useful concept for two reasons: It occurs in the real world and it is a powerful tool for implementing complex systems. Metadata can be confusing to model, because it blurs the distinction between descriptor and referent. Reification, the promotion of something that is not an object into an object, can be a helpful technique for meta applications.

Explicit constraints on classes, associations, and attributes can increase the precision of a model. Generalization and multiplicity are examples of constraints built into the fabric of class modeling. Derived elements may appear in a model but do not add fundamental information.