functionality. Such an approach might seem the most direct way of implementing a desired goal, but the resulting system can be fragile. If the requirements change, a system based on decomposing functionality may require massive restructuring. (To be fair, these methodologies are more complex than this. See Chapter 12 for more details.)

By contrast, the object-oriented approach focuses first on identifying objects from the application domain, then fitting procedures around them. Although this may seem more indirect, object-oriented software holds up better as requirements evolve, because it is based on the underlying framework of the application domain itself, rather than the ad-hoc functional requirements of a single problem.

1.3 OBJECT-ORIENTED THEMES

There are several themes underlying object-oriented technology. Although these themes are not unique to object-oriented systems, they are particularly well supported in object-oriented systems.

1.3.1 Abstraction

Abstraction consists of focusing on the essential, inherent aspects of an entity and ignoring its accidental properties. In system development, this means focusing on what an object is and does, before deciding how it should be implemented. Use of abstraction preserves the freedom to make decisions as long as possible by avoiding premature commitments to details. Most modern languages provide data abstraction, but the ability to use inheritance and polymorphism provides additional power. Use of abstraction during analysis means dealing only with application-domain concepts, not making design and implementation decisions before the problem is understood. Proper use of abstraction allows the same model to be used for analysis, high-level design, program structure, database structure, and documentation. A language-independent style of design defers programming details until the final, relatively mechanical stage of development.

1.3.2 Encapsulation

Encapsulation (also *information hiding*) consists of separating the external aspects of an object, which are accessible to other objects, from the internal implementation details of the object, which are hidden from other objects. Encapsulation prevents a program from becoming so interdependent that a small change has massive ripple effects. The implementation of an object can be changed without affecting the applications that use it. One may want to change the implementation of an object to improve performance, fix a bug, consolidate code, or for porting. Encapsulation is not unique to object-oriented languages, but the ability to combine data structure and behavior in a single entity makes encapsulation cleaner and more powerful than in conventional languages that separate data structure and behavior.

1.3.3 Combining Data and Behavior

The caller of an operation need not consider how many implementations of a given operation exist. Operator polymorphism shifts the burden of deciding what implementation to use from the calling code to the class hierarchy. For example, non-object-oriented code to display the contents of a window must distinguish the type of each figure, such as polygon, circle, or text, and call the appropriate procedure to display it. An object-oriented program would simply invoke the *draw* operation on each figure; the decision of which procedure to use is made implicitly by each object, based on its class. It is unnecessary to repeat the choice of procedure every time the operation is called in the application program. Maintenance is easier, because the calling code need not be modified when a new class is added. In an object-oriented system, the data structure hierarchy is identical to the operation inheritance hierarchy (Figure 1.3).

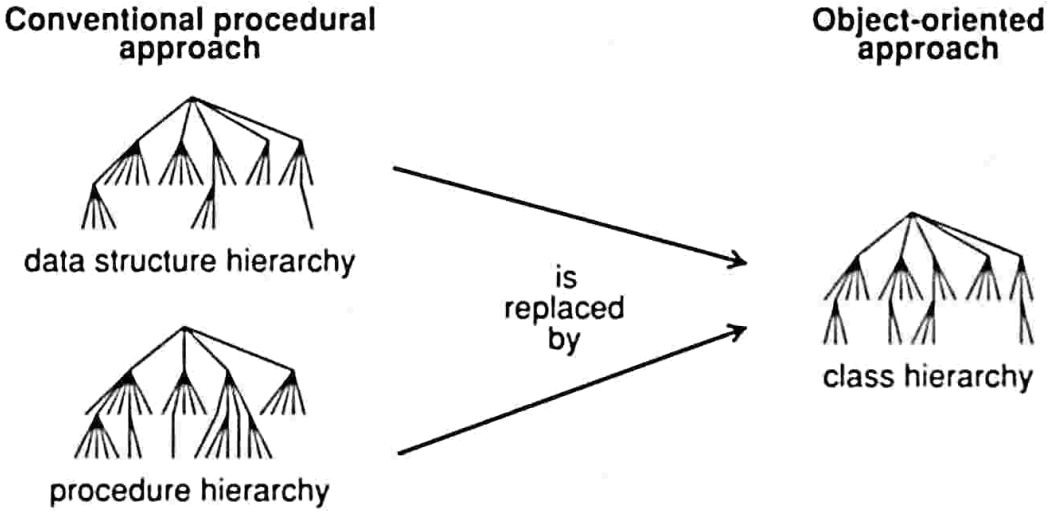


Figure 1.3 An object-oriented approach has one unified hierarchy

1.3.4 Sharing

Object-oriented techniques promote sharing at several different levels. Inheritance of both data structure and behavior allows common structure to be shared among several similar subclasses without redundancy. The sharing of code using inheritance is one of the main advantages of object-oriented languages. More important than the savings in code is the conceptual clarity from recognizing that different operations are all really the same thing. This reduces the number of distinct cases that must be understood and analyzed.

Object-oriented development not only allows information to be shared within an application, but also offers the prospect of reusing designs and code on future projects. Although this possibility has been overemphasized as a justification for object-oriented technology, object-oriented development provides the tools, such as abstraction, encapsulation, and inheritance, to build libraries of reusable components. Object-orientation is not a magic formula to ensure reusability, however. Reuse does not just happen; it must be planned by thinking beyond the immediate application and investing extra effort in a more general design.

1.3.5 Emphasis on Object Structure, Not Procedure Structure

Object-oriented technology stresses specifying what an object *is*, rather than how it is *used*. The uses of an object depend highly on the details of the application and frequently change during development. As requirements evolve, the features supplied by an object are much more stable than the ways it is used, hence software systems built on object structure are more stable in the long run [Booch-86]. Object-oriented development places a greater emphasis on data structure and a lesser emphasis on procedure structure than traditional functional-decomposition methodologies. In this respect, object-oriented development is similar to information modeling techniques used in database design, although object-oriented development adds the concept of class-dependent behavior.

1.3.6 Synergy

Identity, classification, polymorphism, and inheritance characterize mainstream object-oriented languages. Each of these concepts can be used in isolation, but together they complement each other synergistically. The benefits of an object-oriented approach are greater than they might seem at first. The greater emphasis on the essential properties of an object forces the software developer to think more carefully and more deeply about what an object is and does, with the result that the system is usually cleaner, more general, and more robust than it would be if the emphasis were only on the use of data and operations. According to Thomas, these various features come together to create a different style of programming [Thomas-89]. Cox claims that encapsulation is the foundation for the object-oriented approach, shifting emphasis from coding technique to packaging, while inheritance builds on encapsulation to make reuse of code practical [Cox-86].

1.4 EVIDENCE FOR USEFULNESS OF OBJECT-ORIENTED DEVELOPMENT

We have been actively using object-oriented development in internal applications at the General Electric Research and Development Center (GE R&D). We have used object-oriented techniques for developing compilers (Chapter 18), graphics (Chapter 19), user interfaces (Chapter 20), databases [Blaha-89], an object-oriented language [Shah-89], CAD systems, simulations, meta models, control systems, and other applications. We have used object-oriented models to document existing programs that are ill-structured and difficult to understand. Our implementation targets have ranged from object-oriented languages to non-object-oriented languages to relational databases. We have successfully taught this approach to others and have used it to communicate with application experts.

We are enthusiastic supporters of object-oriented development and see no reason it should not be used on most software projects. The main benefit is not reduced development time; object-oriented development may take more time than conventional development, because it is intended to promote future reuse and reduce downstream errors and maintenance. The time until code is first completed is probably about the same as, or slightly greater than, using a conventional approach. However, subsequent iterations of an object-oriented devel-

PART 1: MODELING CONCEPTS

2

Modeling as a Design Technique

A model is an abstraction of something for the purpose of understanding it before building it. Because a model omits nonessential details, it is easier to manipulate than the original entity. Abstraction is a fundamental human capability that permits us to deal with complexity. Engineers, artists, and craftsmen have built models for thousands of years to try out designs before executing them. Development of hardware and software systems is no exception. To build complex systems, the developer must abstract different views of the system, build models using precise notations, verify that the models satisfy the requirements of the system, and gradually add detail to transform the models into an implementation.

Part 1 of the book describes the concepts and notations involved in object-oriented modeling. These concepts are applied to analysis, design, and implementation in Parts 2 and 3 of the book. This chapter discusses modeling in general and then introduces the three kinds of object-oriented models composing the Object Modeling Technique: the object model, which describes static structure; the dynamic model, which describes temporal relationships; and the functional model, which describes functional relationships among values.

2.1 MODELING

Designers build many kinds of models for various purposes before constructing things. Examples include architectural models to show customers, airplane scale models for wind-tunnel tests, pencil sketches for composition of oil paintings, blueprints of machine parts, storyboards of advertisements, and outlines of books. Models serve several purposes:

• *Testing a physical entity before building it.* The medieval masons did not know modern physics, but they built scale models of the Gothic cathedrals to test the forces on the structure. Scale models of airplanes, cars, and boats have been tested in wind tunnels and water

tanks to improve their aerodynamics. Recent advances in computation permit the simulation of many physical structures without having to build physical models. Not only is simulation cheaper, but it provides information that is too fleeting or inaccessible to be measured from a physical model. Both physical models and computer models are usually cheaper than building a complete system and enable flaws to be corrected early.

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

- *Visualization.* Storyboards of movies, television shows, and advertisements allow the writers to see how their ideas flow. Awkward transitions, dangling ends, and unnecessary segments can be modified before detailed writing begins. Artists' sketches allow them to block out their ideas and make changes before committing them to oil or stone.

- *Reduction of complexity.* Perhaps the main reason for modeling, which incorporates all the previous reasons, is to deal with systems that are too complex to understand directly. The human mind can cope with only a limited amount of information at one time. Models reduce complexity by separating out a small number of important things to deal with at a time.

2.1.1 Abstraction

Abstraction is the selective examination of certain aspects of a problem. The goal of abstraction is to isolate those aspects that are important for some purpose and suppress those aspects that are unimportant. Abstraction must always be for some purpose, because the purpose determines what is and is not important. Many different abstractions of the same thing are possible, depending on the purpose for which they are made.

All abstractions are incomplete and inaccurate. Reality is a seamless web. Anything we say about it, any description of it, is an abridgement. All human words and language are abstractions—incomplete descriptions of the real world. This does not destroy their usefulness. The purpose of an abstraction is to limit the universe so we can do things. In building models, therefore, you must not search for absolute truth but for adequacy for some purpose. There is no single "correct" model of a situation, only adequate and inadequate ones.

A good model captures the crucial aspects of a problem and omits the others. Most computer languages, for example, are poor vehicles for modeling algorithms because they force the specification of implementation details that are irrelevant to the algorithm. A model that contains extraneous detail unnecessarily limits your choice of design decisions and diverts attention from the real issues.

2.2 THE OBJECT MODELING TECHNIQUE

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

Modeling Technique (OMT) is our name for the methodology that combines these three views of modeling systems. The *object model* represents the static, structural, "data" aspects of a system. The *dynamic model* represents the temporal, behavioral, "control" aspects of a system. The *functional model* represents the transformational, "function" aspects of a system. A typical software procedure incorporates all three aspects: It uses data structures (object model), it sequences operations in time (dynamic model), and it transforms values (functional model). Each model contains references to entities in other models. For example, operations are attached to objects in the object model but more fully expanded in the functional model.

The three kinds of models separate a system into orthogonal views that can be represented and manipulated with a uniform notation. The different models are not completely independent—a system is more than a collection of independent parts—but each model can be examined and understood by itself to a large extent. The interconnections between the different models are limited and explicit. Of course, it is always possible to create bad designs in which the three models are so intertwined that they cannot be separated, but a good design isolates the different aspects of a system and limits the coupling between them.

Each of the three models evolves during the development cycle. During analysis, a model of the application domain is constructed without regard for eventual implementation. During design, solution-domain constructs are added to the model. During implementation, both application-domain and solution-domain constructs are coded. The word *model* has two dimensions—a view of a system (object model, dynamic model, or functional model) and a stage of development (analysis, design, or implementation). The meaning is generally clear from context.

2.2.1 Object Model

The *object model* describes the structure of objects in a system—their identity, their relationships to other objects, their attributes, and their operations. The object model provides the essential framework into which the dynamic and functional models can be placed. Changes and transformations are meaningless unless there is something to be changed or transformed. Objects are the units into which we divide the world, the molecules of our models.

Our goal in constructing an object model is to capture those concepts from the real world that are important to an application. In modeling an engineering problem, the object model should contain terms familiar to engineers; in modeling a business problem, terms from the business; in modeling a user interface, terms from the application domain. An analysis model should not contain computer constructs unless the application being modeled is inherently a computer problem, such as a compiler or an operating system. The design model describes how to solve a problem and may contain computer constructs.

The object model is represented graphically with object diagrams containing object classes. Classes are arranged into hierarchies sharing common structure and behavior and are associated with other classes. Classes define the attribute values carried by each object instance and the operations which each object performs or undergoes.

2.2.2 Dynamic Model

The *dynamic model* describes those aspects of a system concerned with time and the sequencing of operations—events that mark changes, sequences of events, states that define the context for events, and the organization of events and states. The dynamic model captures *control*, that aspect of a system that describes the sequences of operations that occur, without regard for what the operations do, what they operate on, or how they are implemented.

The dynamic model is represented graphically with state diagrams. Each state diagram shows the state and event sequences permitted in a system for one class of objects. State diagrams also refer to the other models. Actions in the state diagrams correspond to functions from the functional model; events in a state diagram become operations on objects in the object model.

2.2.3 Functional Model

The *functional model* describes those aspects of a system concerned with transformations of values—functions, mappings, constraints, and functional dependencies. The functional model captures what a system does, without regard for how or when it is done.

The functional model is represented with data flow diagrams. Data flow diagrams show the dependencies between values and the computation of output values from input values and functions, without regard for when or if the functions are executed. Traditional computing concepts such as expression trees are examples of functional models, as are less traditional concepts such as spreadsheets. Functions are invoked as actions in the dynamic model and are shown as operations on objects in the object model.

2.2.4 Relationship among Models

Each model describes one aspect of the system but contains references to the other models. The object model describes data structure that the dynamic and functional models operate on. The operations in the object model correspond to events in the dynamic model and functions in the functional model. The dynamic model describes the control structure of objects. It shows decisions which depend on object values and which cause actions that change object values and invoke functions. The functional model describes functions invoked by operations in the object model and actions in the dynamic model. Functions operate on data values specified by the object model. The functional model also shows constraints on object values.

There are occasional ambiguities about which model should contain a piece of information. This is natural, because any abstraction is only a rough cut at reality; something will inevitably straddle the boundaries. Some properties of a system may be poorly represented by the models. This is also normal, because no abstraction can capture everything; the goal is to simplify the system description without loading down the model with so many constructs that it becomes a burden and not a help. For those things that the model does not adequately capture, natural language or application-specific notation is still a perfectly acceptable tool.

3

Object Modeling

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

Chapter 3 discusses basic object modeling concepts that will be used throughout the book. For each concept, we discuss the logical meaning, present the corresponding OMT 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 OBJECTS AND CLASSES

3.1.1 Objects

The purpose of object modeling is to describe objects. For example, *Joe Smith*, *Simplex company*, *Lassie*, *process number 7648*, and *the top window* are objects. An object is simply something that makes sense in an application context.

We define an *object* as a concept, abstraction, or thing with crisp boundaries and meaning for the problem at hand. Objects serve two purposes: They promote understanding of the real world and provide a practical basis for computer implementation. Decomposition of a problem into objects depends on judgment and the nature of the problem. There is no one correct representation.

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.

The word *object* is often vaguely used in the literature. Sometimes *object* means a single thing, other times it refers to a group of similar things. Usually the context resolves any ambiguity. When we want to be precise and refer to exactly one thing, we will use the phrase *object instance*. We will use the phrase *object class* to refer to a group of similar things.

3.1.2 Classes

An *object class* describes a group of objects with similar properties (attributes), common behavior (operations), common relationships to other objects, and common semantics. *Person*, *company*, *animal*, *process*, and *window* are all object classes. Each person has an age, IQ, and may work at a job. Each process has an owner, priority, and list of required resources. Objects and object classes often appear as nouns in problem descriptions.

The abbreviation *class* is often used instead of *object class*. Objects in a class have the same attributes and behavior patterns. Most objects derive their individuality from differences in their attribute values and relationships to other objects. However, objects with identical attribute values and relationships are possible.

The objects in a class share a common semantic purpose, above and beyond the requirement of common attributes and behavior. Thus even though a barn and a horse both have a cost and age, they may belong to different classes. If barn and horse were regarded as purely financial assets, they may 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 object-oriented 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 object 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. Operations can be written 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, or 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 Object Diagrams

We began this chapter by discussing some basic modeling concepts, specifically *object* and *class*. We have described these concepts with examples and prose. Since this approach is

vague for more complex topics, we need a formalism for expressing object models that is coherent, precise, and easy to formulate.

Object diagrams provide a formal graphic notation for modeling objects, classes, and their relationships to one another. Object diagrams are useful both for abstract modeling and for designing actual programs. Object diagrams are concise, easy to understand, and work well in practice. We use object diagrams throughout this book. New concepts are illustrated by object diagrams to introduce the notation and clarify our explanation of concepts. There are two types of object diagrams: class diagrams and instance diagrams.

A *class diagram* is a schema, pattern, or template for describing many possible instances of data. A class diagram describes object classes.

An *instance diagram* describes how a particular set of objects relate to each other. An instance diagram describes object instances. Instance diagrams are useful for documenting test cases (especially scenarios) and discussing examples. A given class diagram corresponds to an infinite set of instance diagrams.

Figure 3.1 shows a class diagram (left) and one possible instance diagram (right) described by it. Objects *Joe Smith*, *Mary Sharp*, and an anonymous person are instances of class *Person*. The OMT symbol for an object instance is a rounded box. The class name in parentheses is at the top of the object box in boldface. Object names are listed in normal font. The OMT symbol for a class is a box with class name in boldface.

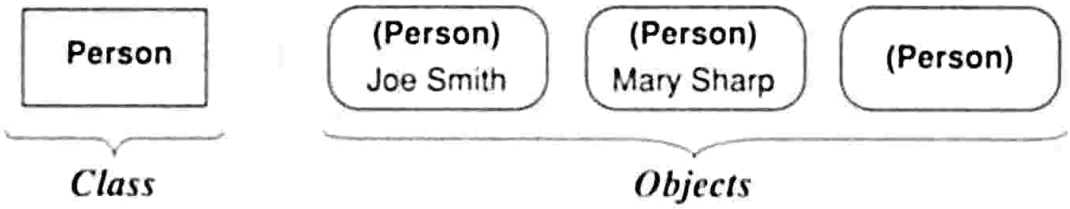


Figure 3.1 Class and objects

Class diagrams describe the general case in modeling a system. Instance diagrams are used mainly to show examples to help to clarify a complex class diagram. The distinction between class diagrams and instance diagrams is in fact artificial; classes and instances can appear on the same object diagram, but in general it is not useful to mix classes and instances. (The exception is metadata, discussed in Section 4.5.)

3.1.4 Attributes

An *attribute* is a data value held by the objects in a class. *Name*, *age*, and *weight* are attributes of *Person* objects. *Color*, *weight*, and *model-year* are attributes of *Car* objects. Each attribute has a value for each object instance. For example, attribute *age* has value "24" in object *Joe Smith*. Paraphrasing, Joe Smith is 24 years old. Different object instances 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 *Company* may each have an attribute called *address*.

An attribute should be a pure data value, not an object. Unlike objects, pure data values do not have 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). The capital of Canada is a city object and should not be modeled as an attribute, but rather as an association between a country object and a city object (explained in Section 3.2). The *name* of this city object is "Ottawa" (the string).

Attributes are listed in the second part of the class box. Each attribute name may be followed by optional details, such as type and default value. The type is preceded by a colon. The default value is preceded by an equal sign. At times, you may choose to omit showing attributes in class boxes. It depends on the level of detail desired in the object model. Class boxes have a line drawn between the class name and attributes. Object boxes do not have this line in order to further differentiate them from class boxes.

Figure 3.2 shows object modeling notation. Class *Person* has attributes *name* and *age*. *Name* is a string and *age* is an integer. One object in class *Person* has the value *Joe Smith* for name and the value *24* for age. Another object has name *Mary Sharp* and age *52*.

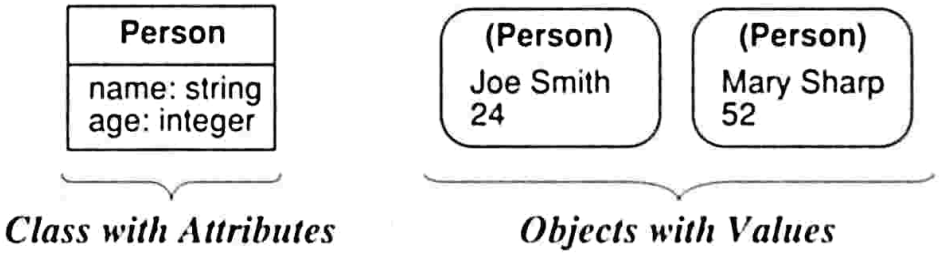


Figure 3.2 Attributes and values

Some implementation media, such as many databases, require an object to have a unique identifier that identifies each object. Explicit object identifiers are not required in an object model. Each object has its own unique identity. Most object-oriented languages automatically generate implicit identifiers with which to reference objects. You need not and should not explicitly list identifiers. Figure 3.3 emphasizes this point. Identifiers are a computer artifact and have no intrinsic meaning beyond identifying an object.

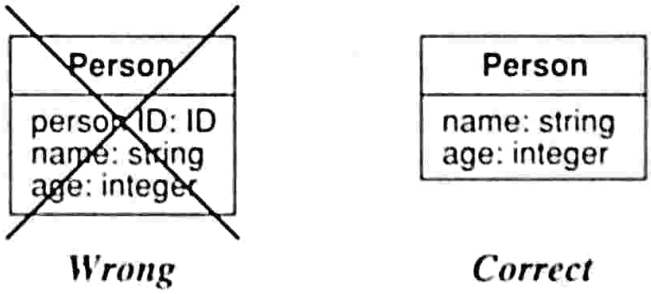


Figure 3.3 Do not explicitly list object identifiers

Do not confuse internal identifiers with real-world attributes. Internal identifiers are purely an implementation convenience and have no meaning in the problem domain. For example, social security number, license plate number, and telephone number are not internal

identifiers because they have meaning in the real world. Social security number, license plate number, and telephone number are legitimate attributes.

3.1.5 Operations and Methods

An *operation* is a function or transformation that may be applied to or by objects in a class. *Hire*, *fire*, and *pay-dividend* 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*. Different methods could be implemented 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, each method may be implemented by a different piece of code.

An operation may have arguments in addition to its target object. Such arguments parameterize the operation but do not affect the choice of method. The method depends only on the class of the target object. (A few object-oriented 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 *file-name* as an argument for one method and *file-pointer* 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 very 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.

Operations are listed in the lower third of the class box. Each operation name may be followed by optional details, such as argument list and result type. An argument list is written in parentheses following the name; the arguments are separated by commas. The name and type of each argument may be given. The result type is preceded by a colon and should not be omitted, because it is important to distinguish operations that return values from those that do not. An empty argument list in parentheses shows explicitly that there are no arguments; otherwise no conclusions can be drawn. Operations may be omitted from high-level diagrams.

In Figure 3.4, the class *Person* has attributes *name* and *age* and operations *change-job* and *change-address*. *Name*, *age*, *change-job*, and *change-address* are features of *Person*.

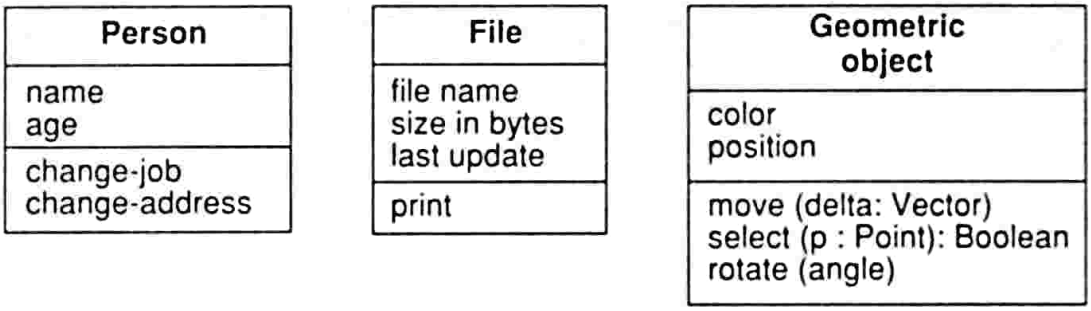


Figure 3.4 Operations

Feature is a generic word for either an attribute or operation. Similarly, *File* has a *print* operation. *Geometric object* 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*.

During modeling, it is useful to distinguish operations that have side effects from those that merely compute a functional value without modifying any objects. The latter kind of operation is called a *query*. Queries with no arguments except the target object may be regarded as derived attributes. For example, the width of a box can be computed from the positions of its sides. A derived attribute is like an attribute in that it is a property of the object itself, and computing it does not change the state of the object. In many cases, an object has a set of attributes whose values are interrelated, of which only a fixed number of values can be chosen independently. An object 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. The remaining attributes may be omitted or may be shown as derived attributes as described in Section 4.7.4.

3.1.6 Summary of Notation for Object Classes

Figure 3.5 summarizes object modeling notation for classes. A class is represented by a box which may have as many as three regions. The regions contain, from top to bottom: class name, list of attributes, and list of operations. Each attribute name may be followed by optional details such as type and default value. Each operation name may be followed by optional details such as argument list and result type. Attributes and operations may or may not be shown; it depends on the level of detail desired.

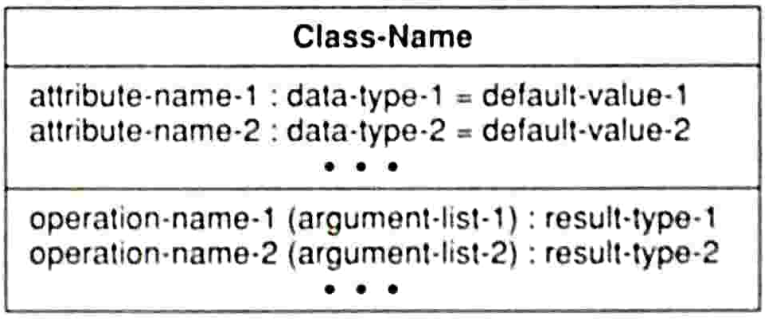


Figure 3.5 Summary of object modeling notation for classes

3.2 LINKS AND ASSOCIATIONS

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

3.2.1 General Concepts

A *link* is a physical or conceptual connection between object instances. For example, Joe Smith *Works-for* Simplex company. Mathematically, a link is defined as a tuple, that is, an ordered list of object instances. A link is an instance of an association.

An *association* describes a group of links with common structure and common semantics. For example, a person *Works-for* a company. All the links in an association connect objects from the same classes. Associations and links often appear as verbs in a problem statement. An association describes a set of potential links in the same way that a class describes a set of potential objects.

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. The direction implied by the name is the *forward* direction; the opposite direction is the *inverse* direction. For example, *Works-for* connects a person to a company. The inverse of *Works-for* could be called *Employs*, and 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 which establish a direction.

Associations are often implemented in programming languages as pointers from one object to another. A *pointer* is an attribute in one object that contains an explicit reference to another object. For example, a data structure for *Person* might contain an attribute *employer* that points to a *Company* object, and a *Company* object might contain an attribute *employees* that points to a set of *Employee* objects. Implementing associations as pointers is perfectly acceptable, but associations should not be modeled this way.

A link shows a relationship between two (or more) objects. Modeling a link as a pointer 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 pointers, such as the pointer from *Person* to *Company* and the pointer from *Company* to a set of *Employee*, hides the fact that the forward and inverse pointers are dependent on each other. All connections among classes should therefore be modeled as associations, even in designs for programs. We must stress that associations are not just database constructs, although relational databases are built on the concept of associations.

Although associations are modeled as bidirectional they do not have to be implemented in both directions. Associations can easily be implemented as pointers if they are only traversed in a single direction. Chapter 10 discusses some trade-offs to consider when implementing associations.

Figure 3.6 shows a one-to-one association and corresponding links. Each association in the class diagram corresponds to a set of links in the instance diagram, just as each class corresponds to a set of objects. Each country has a capital city. *Has-capital* is the name of the

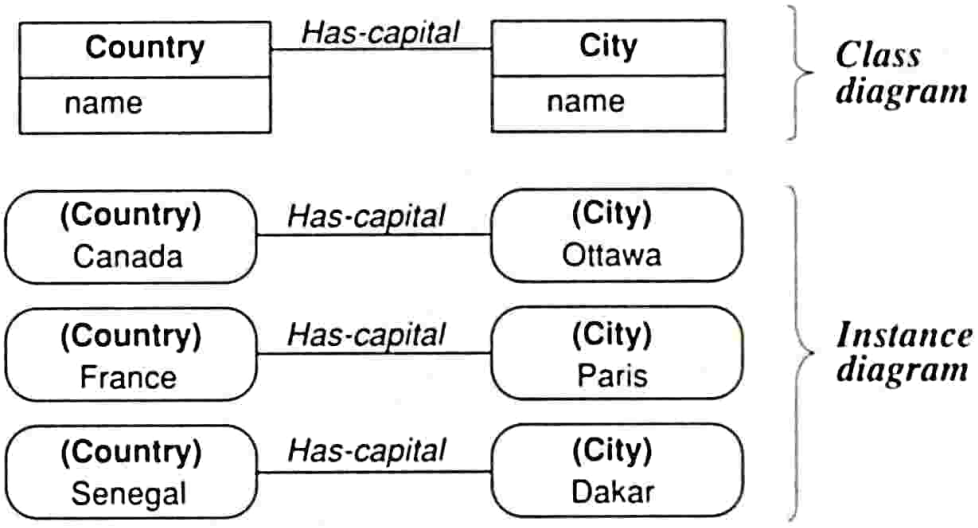


Figure 3.6 One-to-one association and links

association. The OMT notation for an association is a line between classes. A link is drawn as a line between objects. Association names are italicized. An association name may be omitted if a pair of classes has a single association whose meaning is obvious. It is good to arrange the classes to read from left-to-right, if possible.

Figure 3.7 is a fragment of an object model for a program. A common task that arises in computer-aided design (CAD) applications is to find connectivity networks: Given a line, find all intersecting lines; given an intersection point, find all lines that pass through it; given an area on the screen, find all intersection points. (We use the word *line* here to mean a finite line segment.)

In the class diagram, each point denotes the intersection of two or more lines; each line has zero or more intersection points. The instance diagram shows one possible set of lines. Lines *L1*, *L2*, and *L3* intersect at point *P1*. Lines *L3* and *L4* intersect at point *P2*. Line *L5* has no intersection points and thus has no link. The solid balls and "2+" are multiplicity symbols. Multiplicity specifies how many instances of one class may relate to each instance of another class and is discussed in the next section.

Associations may be binary, ternary, or higher order. In practice, the vast majority are binary or qualified (a special form of ternary discussed later). We have encountered a few general ternary and few, if any, of order four or more. Higher order associations are more complicated to draw, implement, and think about than binary associations and should be avoided if possible.

Figure 3.8 shows a *ternary* association: Persons who are programmers use computer languages on projects. This ternary 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 OMT symbol for general ternary and n-ary associations is a diamond with lines connecting to related classes. The name of the association is written next to the diamond. Note that we did not name the association or links in Figure 3.8. Association names are optional and a matter of modeling judgment. Associations are often left unnamed when they can be easily identified by their classes. (This convention does not work if there are multiple associations between the same classes.)

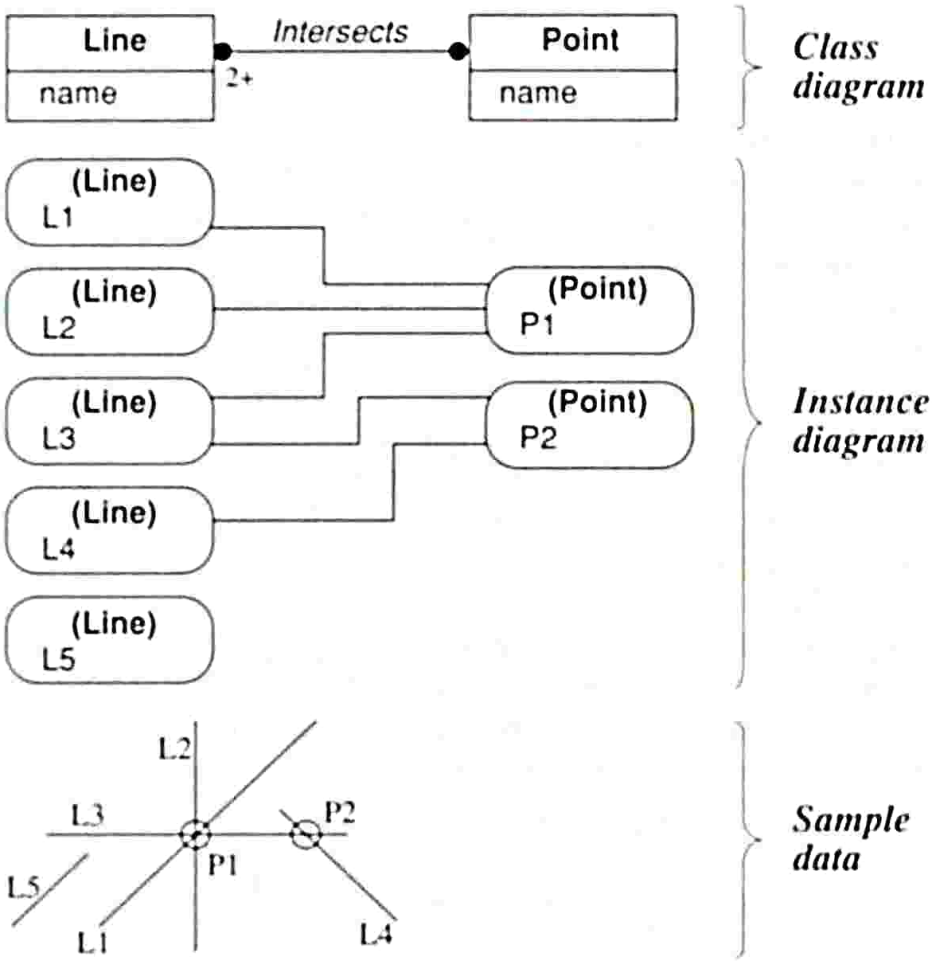


Figure 3.7 Many-to-many association and links

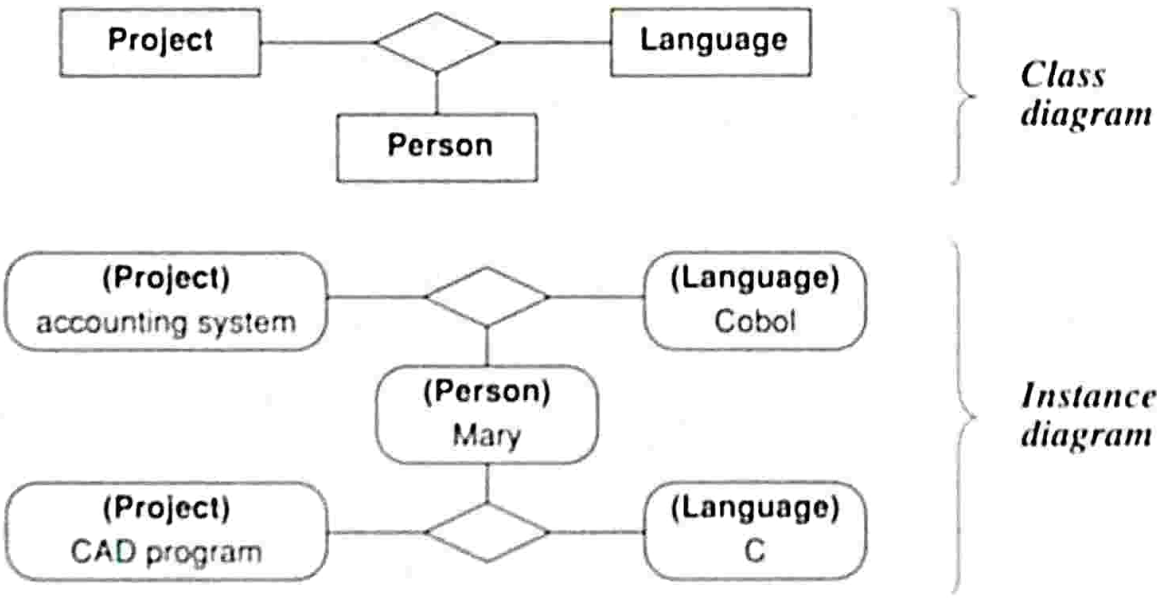


Figure 3.8 Ternary association and links

3.2.2 Multiplicity

Multiplicity specifies how many instances of one class may relate to a single instance of an associated class. Multiplicity constrains the number of related objects. Multiplicity is often described as being "one" or "many," but more generally it is a (possibly infinite) subset of the non-negative integers. Generally the multiplicity value is a single interval, but it may be a set of disconnected intervals. For example, the number of doors on a sedan is 2 or 4. Object diagrams indicate multiplicity with special symbols at the ends of association lines. In the most general case, multiplicity can be specified with a number or set of intervals, such as "1" (exactly one), "1+" (one or more), "3-5" (three to five, inclusive), and "2,4,18" (two, four, or eighteen). There are special line terminators to indicate certain common multiplicity values. A solid ball is the OMT symbol for "many," meaning zero or more. A hollow ball indicates "optional," meaning zero or one. A line without multiplicity symbols indicates a one-to-one association. In the general case, the multiplicity is written next to the end of the line, for example, "1+" to indicate one or more.

Reviewing our past examples, Figure 3.6 illustrates one-to-one multiplicity. Each country has one capital city. A capital city administers one country. (In fact, some countries, such as 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.)

The association in Figure 3.7 exhibits many-to-many multiplicity. A line may have zero or more intersection points. An intersection point may be associated with two or more lines. In this particular case, *L1*, *L2*, and *L4* have one intersection point; *L3* has two intersection points; *L5* has no intersection points. *P1* intersects with three lines; *P2* intersects with two lines.

Figure 3.9 illustrates zero-or-one, or optional, 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 a role name, discussed in Section 3.3.3.)

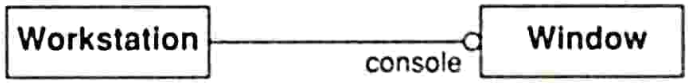


Figure 3.9 Zero-or-one multiplicity

Multiplicity depends on assumptions and how you define the boundaries of a problem. Vague requirements often make multiplicity uncertain. You should not worry excessively about multiplicity early in software development. First determine objects, classes, and associations, then decide on multiplicity.

Determining multiplicity often exposes hidden assumptions built into the model. For example, is the *Works-for* 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, an auto workers' union maintaining member records may consider second jobs irrelevant. Explicitly representing a model with object diagrams helps 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 phone number utility programs are unable to accommodate persons with multiple phone numbers. On the other hand, overestimating multiplicity imposes extra 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.

This chapter only considers multiplicity for binary associations. The solid and hollow ball notation is ambiguous for n-ary (n > 2) associations, for which multiplicity is a more complex topic. Section 4.6 extends our treatment of multiplicity to n-ary associations.

3.2.3 The Importance of Associations

The notion of an association is certainly not a new concept. Associations have been widely used throughout the database modeling community for years. (See Chapter 12 for details.) In contrast, few programming languages explicitly support associations. We nevertheless emphasize that associations are a useful modeling construct for programs as well as databases and real-world systems, regardless of how they are implemented. During conceptual modeling, you should not bury pointers or other object references inside objects as attributes. Instead you should model them as associations to indicate that the information they contain is not subordinate to a single class, but depends on two or more classes [Rumbaugh-87].*

Some object-oriented authors feel that every piece of information should be attached to a single class, and they argue that associations violate encapsulation of information into classes. We do not agree with this viewpoint. Some information inherently transcends a single class, and the failure to treat associations on an equal footing with classes can lead to programs containing hidden assumptions and dependencies.

Most object-oriented languages implement associations with object pointers. Pointers can be regarded as an implementation optimization introduced during the later stages of design. It is also possible to implement association objects directly, but the use of association objects during implementation is really a design decision (see Chapter 10).

3.3 ADVANCED LINK AND ASSOCIATION CONCEPTS

3.3.1 Link Attributes

An attribute is a property of the objects in a class. Similarly, a *link attribute* is a property of the links in an association. In Figure 3.10, *access permission* is an attribute of *Accessible by*. Each link attribute has a value for each link, as illustrated by the sample data at the bottom

---

* The term *association* as used in this book is synonymous with the term *relation* used in [Rumbaugh-87] and with the use of the term *relation* as used in discrete mathematics. We have used the term *association* to avoid confusion with the more restricted use of the term *relation* as used in relational databases, which usually permit relations between pure values only, not between objects with identity.

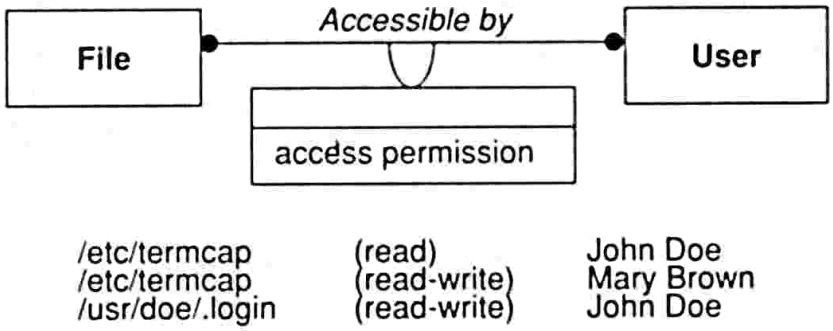


Figure 3.10 Link attribute for a many-to-many association

of the figure. The OMT notation for a link attribute is a box attached to the association by a loop; one or more link attributes may appear in the second region of the box. This notation emphasizes the similarity between attributes for objects and attributes for links.

Many-to-many associations provide the most compelling rationale for link attributes. Such an attribute is unmistakably a property of the link and cannot be attached to either object. In Figure 3.10, *access permission* is a joint property of *File* and *User*, and cannot be attached to either *File* or *User* alone without losing information.

Figure 3.11 presents link attributes for two many-to-one associations. Each person working for a company receives a salary and has a job title. The boss evaluates the performance of each worker. Link attributes may also occur for one-to-one associations.

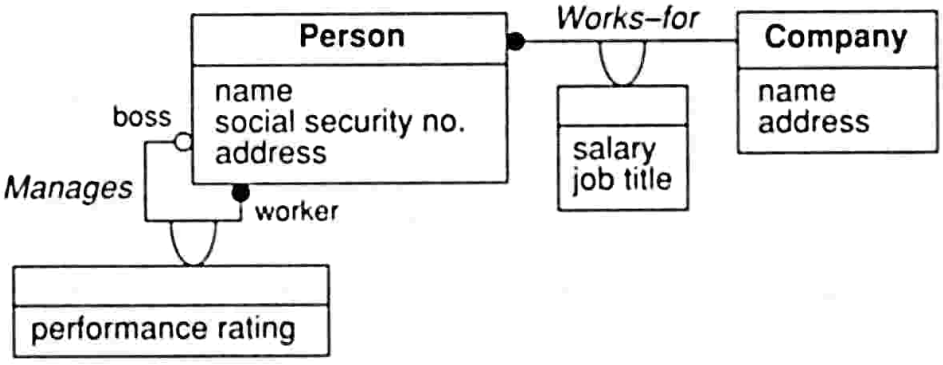
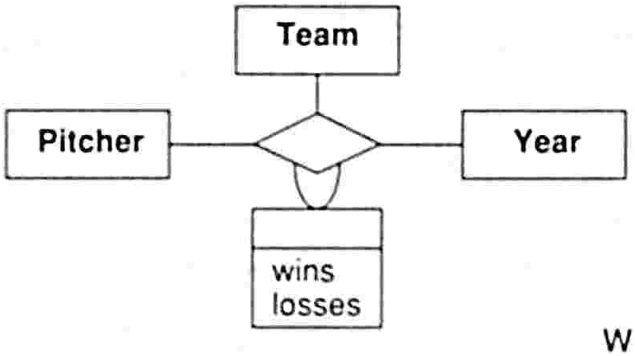


Figure 3.11 Link attributes for one-to-many associations

Figure 3.12 shows link attributes for a ternary association. A pitcher may play for many teams in a given year. A pitcher may also play many years for the same team. Each team has many pitchers. For each combination of team and year, a pitcher has a won-loss record. Thus for instance, Harry Eisenstat pitched for the Cleveland Indians in 1939, winning 6 games and losing 7 games.

Figure 3.13 shows how it is possible to fold link attributes for one-to-one and one-to-many associations into the class opposite the "one" side. This is not possible for many-to-many associations. As a rule, link attributes should not be folded into a class because future flexibility is reduced if the multiplicity of the association should change. Either form in Figure 3.13 can express a one-to-many association. However, only the link attribute form remains correct if the multiplicity of *Works-for* is changed to many-to-many.



			W	L
Harry Eisenstat	Cleveland Indians	1939	6	7
Harry Eisenstat	Detroit Tigers	1939	2	2
Willis Hudlin	Cleveland Indians	1939	9	10
Willis Hudlin	Cleveland Indians	1940	2	1
Willis Hudlin	Washington Senators	1940	1	2
Willis Hudlin	St. Louis Browns	1940	0	1

Figure 3.12 Link attributes for a ternary association

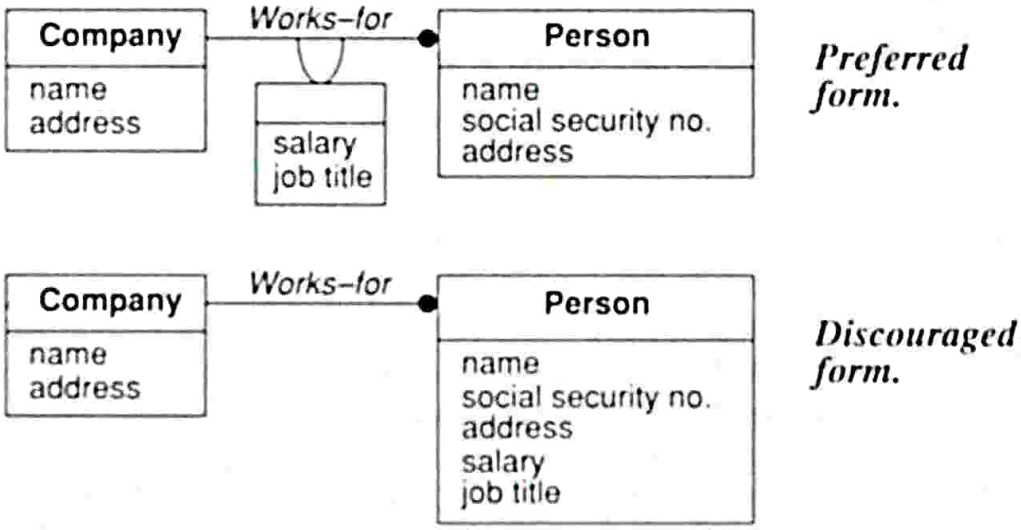


Figure 3.13 Link attribute versus object attribute

3.3.2 Modeling an Association as a Class

Sometimes it is useful to model an association as a class. Each link becomes one instance of the class. The link attribute box introduced in the previous section is actually a special case of an association as a class, and may have a name and operations in addition to attributes. Figure 3.14 shows the authorization information for users on workstations. Users may be authorized on many workstations. Each authorization carries a priority and access privileges, shown as link attributes. A user has a home directory for each authorized workstation, but the same home directory can be shared among several workstations or among several users. The home directory is shown as a many-to-one association between the authorization class and the directory class. It is useful to model an association as a class when links can participate in associations with other objects or when links are subject to operations.

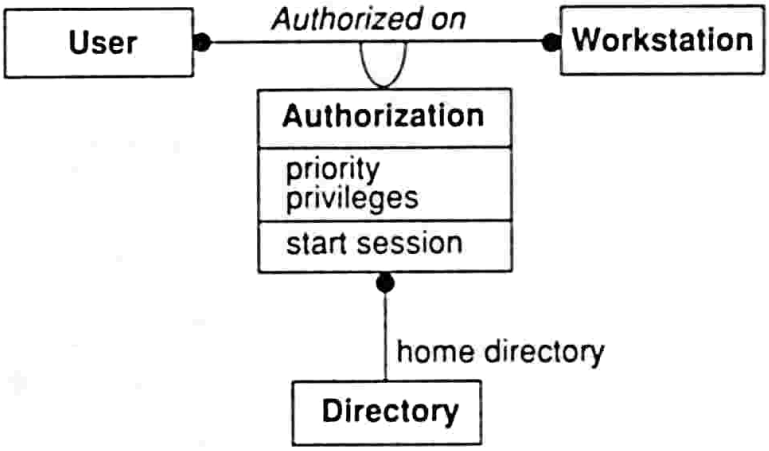
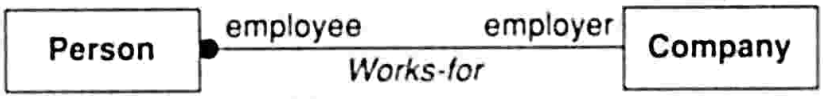


Figure 3.14 Modeling an association as a class

3.3.3 Role Names

A *role* is one end of an association. A binary association has two roles, each of which may have a *role name*. A *role name* is a name that uniquely identifies one end of an association. Roles provide a way of viewing a binary association as a traversal from one object to a set of associated objects. Each role on a binary association identifies an object or set of objects associated with an object at the other end. From the point of view of the object, traversing the association is an operation that yields related objects. The role name is a derived attribute whose value is a set of related objects. Use of role names provides a way of traversing associations from an object at one end, without explicitly mentioning the association. Roles often appear as nouns in problem descriptions.

Figure 3.15 specifies how *Person* and *Company* participate in association *Works-for*. A person assumes the role of *employee* with respect to a company; a company assumes the role of *employer* with respect to a person. A role name is written next to the association line near the class that plays the role (that is, the role name appears on the destination end of the traversal). Use of role names is optional, but it is often easier and less confusing to assign role names instead of, or in addition to, association names.



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

Figure 3.15 Role names for an association

Role names are necessary for associations between two objects of the same class. For example, *boss* and *worker* distinguish the two employees participating in the *Manages* association in Figure 3.11. Role names are also useful to distinguish between two associations between the same pair of classes. When there is only a single association between a pair of

distinct classes, the names of the classes often serve as good role names, in which case the role names may be omitted on the diagram.

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

Figure 3.16 shows both uses of role names. A directory may contain many other directories and may optionally be contained in another directory. Each directory has exactly one user who is an owner, and many users who are authorized to use the directory.

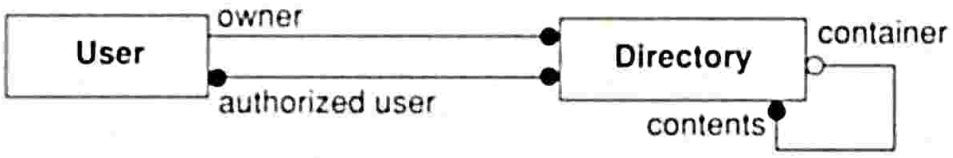


Figure 3.16 Role names for a directory hierarchy

An n-ary association has a role for each end. The role names distinguish the ends of the association and are necessary if a class participates in an n-ary association more than once. Associations of degree 3 or more cannot simply be traversed from one end to another as binary associations can, so the role names do not represent derived attributes of the participating classes. For example, in Figure 3.12, both a team and a year are necessary to obtain a set of pitchers.

3.3.4 Ordering

Usually the objects on the "many" side of an association have no explicit order, and can be regarded as a set. Sometimes, however, the objects are explicitly ordered. For example, Figure 3.17 shows a workstation screen containing a number of overlapping windows. The windows are explicitly ordered, so only the topmost window is visible at any point on the screen. The ordering is an inherent part of the association. An ordered set of objects on the "many" end of an association is indicated by writing "{ordered}" next to the multiplicity dot for the role. This is a special kind of constraint. (See Section 4.7 for a discussion of constraints.)

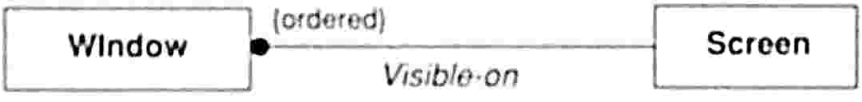


Figure 3.17 Ordered sets in an association

3.3.5 Qualification

A *qualified association* relates two object classes and a *qualifier*. The qualifier is a special attribute that reduces the effective multiplicity of an association. One-to-many and many-to-many associations may be qualified. The qualifier distinguishes among the set of objects at

the many end of an association. A qualified association can also be considered a form of ternary association.

For example, in Figure 3.18 a directory has many files. A file may only belong to a single directory.* Within the context of a directory, the file name specifies a unique file. *Directory* and *File* are object classes and *file name* is the qualifier. A directory plus a file name yields a file. A file corresponds to a directory and a file name. Qualification reduces the effective multiplicity of this association from one-to-many to one-to-one. A directory has many files, each with a unique name.

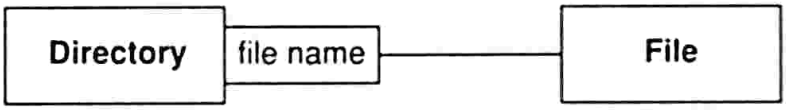


Figure 3.18 A qualified association

Qualification improves semantic accuracy and increases the visibility of navigation paths. It is much more informative to be told that a directory and file name combine to identify a file, rather than be told that a directory has many files. The qualification syntax also indicates that each file name is unique within its directory. One way to find a file is to first find the directory and then traverse the file name link.

A qualifier is drawn as a small box on the end of the association line near the class it qualifies. *Directory + file name* yields a *File*, therefore *file name* is listed in a box contiguous to *Directory*.

Qualification often occurs in real problems, frequently because of the need to supply names. There normally is a context within which a name has meaning. For instance, a directory provides the context for a file name.

Figure 3.19 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. (This may actually not be true for stocks.) The unqualified notation cannot accommodate different ticker symbols for the same company on different exchanges.

Qualification usually reduces multiplicity from many to one, but not always. In Figure 3.20, a company has one president and one treasurer but many persons serving on the board of directors. Qualification partitions a set of related objects into disjoint subsets, but the subsets may contain more than one object.

3.3.6 Aggregation

Aggregation is the "part-whole" or "a-part-of" relationship in which objects representing the *components* of something are associated with an object representing the entire *assembly*. One

* This is only true for some operating systems. For example, a PC-DOS file does belong to a single directory. A UNIX file may belong to multiple directories. Once again, the precise nature of an object model depends upon the application.

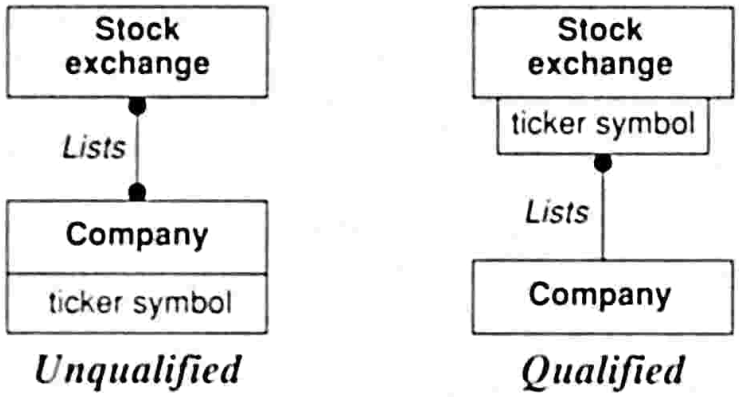
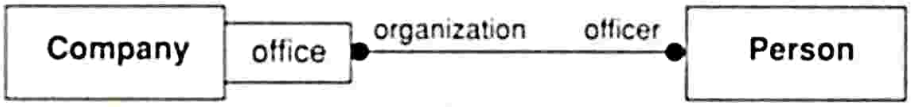


Figure 3.19 Unqualified and qualified association



ABC widgets	President	Roger Slick
ABC widgets	Treasurer	Joe Embezzle
ABC widgets	Director	Joe Doe
ABC widgets	Director	Jane Doe
ABC widgets	Director	Moe Brown
XYZ candy	President	Moe Brown

Figure 3.20 Many-to-many qualification

common example is the bill-of-materials or parts explosion tree. For example, a name, argument list, and a compound statement are part of a C-language function definition, which in turn is part of an entire program. Aggregation is a tightly coupled form of association with some extra semantics. 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. Finally, some properties of the assembly *propagate* to the components as well, possibly with some local modifications. For example, the environment of a statement within a function definition is the same as the environment of the whole function, except for changes made within the function. The speed and location of a door handle is obtained from the door of which it is a part; the door in turn obtains its properties from the car of which it is a part. Unless there are common properties of components that can be attached to the assembly as a whole, there is little point in using aggregation. A parts tree is clearly an aggregation, but there are borderline cases where the use of aggregation is not clear-cut. When in doubt, use ordinary association. Section 4.1 explores the use of aggregation in more detail.

We define an aggregation relationship as relating an assembly class to *one* component class. An assembly with many kinds of components corresponds to many aggregation relationships. We define each individual pairing as an aggregation so that we can specify the multiplicity of each component within the assembly. This definition emphasizes that aggregation is a special form of association.

Aggregation is drawn like association, except a small diamond indicates the assembly end of the relationship. Figure 3.21 shows a portion of an object model for a word processing program. A document consists of many paragraphs, each of which consists of many sentences.

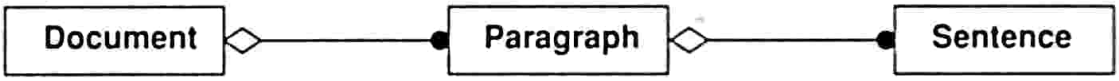


Figure 3.21 Aggregation

The existence of a component object may depend on the existence of the aggregate object of which it is part. For example, a binding is a part of a book. A binding cannot exist apart from a book. In other cases, component objects have an independent existence, such as mechanical parts from a bin.

Figure 3.22 demonstrates that aggregation may have an arbitrary number of levels. A microcomputer is composed of one or more monitors, a system box, an optional mouse, and a keyboard. A system box, in turn, has a chassis, a CPU, many RAM chips, and an optional fan. When we have a collection of components that all belong to the same assembly, we can combine the lines into a single aggregation tree in the diagram. The aggregation tree is just a shorthand notation that is simpler than drawing many lines connecting components to an assembly. An object model should make it easy to visually identify levels in a part hierarchy.

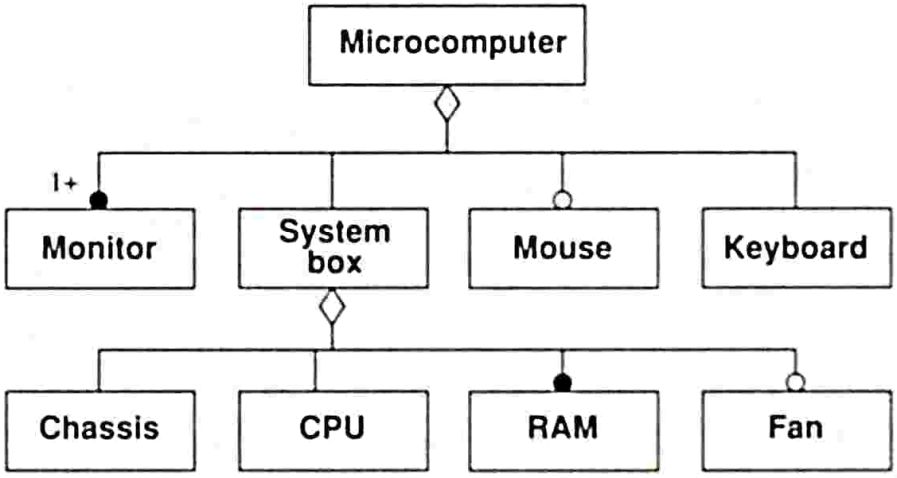


Figure 3.22 Multilevel aggregation

3.4 GENERALIZATION AND INHERITANCE

3.4.1 General Concepts

Generalization and inheritance are powerful abstractions for sharing similarities among classes while preserving their differences. For example, we would like to be able to model the following situation: Each piece of equipment has a manufacturer, weight, and cost.

Pumps also have suction pressure and flow rate. Tanks also have volume and pressure. We would like to define equipment features just once and then add details for pump, tank, and other equipment types.

Generalization is the relationship between a class and one or more refined versions of it. The class being refined is called the *superclass* and each refined version is called a *subclass*. For example, *Equipment* is the superclass of *Pump* and *Tank*. Attributes and operations common to a group of subclasses are attached to the superclass and shared by each subclass. Each subclass is said to *inherit* the features of its superclass. For example, *Pump* inherits attributes *manufacturer*, *weight*, and *cost* from *Equipment*. Generalization is sometimes called the "is-a" relationship because each instance of a subclass is an instance of the superclass as well.

Generalization and inheritance are transitive across an arbitrary number of levels. The terms *ancestor* and *descendent* refer to generalization of classes across multiple levels. An instance of a subclass is simultaneously an instance of all its ancestor classes. The state of an instance includes a value for every attribute of every ancestor class. Any operation on any ancestor class can be applied to an instance. Each subclass not only inherits all the features of its ancestors but adds its own specific attributes and operations as well. For example, *Pump* adds attribute *flow rate*, which is not shared by other kinds of equipment.

The notation for generalization is a triangle connecting a superclass to its subclasses. The superclass is connected by a line to the apex of the triangle. The subclasses are connected by lines to a horizontal bar attached to the base of the triangle. For convenience, the triangle can be inverted, and subclasses can be connected to both the top and bottom of the bar, but if possible the superclass should be drawn on top and the subclasses on the bottom.

Figure 3.23 shows an equipment generalization. Each piece of equipment is a pump, heat exchanger, tank, or another type of equipment. There are several kinds of pumps: centrifugal, diaphragm, and plunger. There are several kinds of tanks: floating roof, pressurized, and spherical. *Pump type* and *tank type* both refine second level generalization classes down to a third level; the fact that the tank generalization symbol is drawn below the pump generalization symbol is not significant. Several object instances 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* assumes the properties of equipment and heat exchanger.

The dangling subclass ellipsis (triple dot) in Figure 3.23 indicates that there are additional subclasses that are not shown on the diagram, perhaps because there is no room on the sheet and they are shown elsewhere, or maybe because enumeration of subclasses is still incomplete.

The words written next to the triangles in the diagram, such as *equipment type*, *pump type*, and *tank type*, are discriminators. A *discriminator* is an attribute of enumeration type that indicates which property of an object is being abstracted by a particular generalization relationship. Only one property should be discriminated at once. For example, class *Vehicle* can be discriminated on propulsion (wind, gas, coal, animal, gravity) and also on operating environment (land, air, water, outer space). The discriminator is simply a name for the basis of generalization. Discriminator values are inherently in one-to-one correspondence with the subclasses of a generalization. For example, the operating environment discriminator for

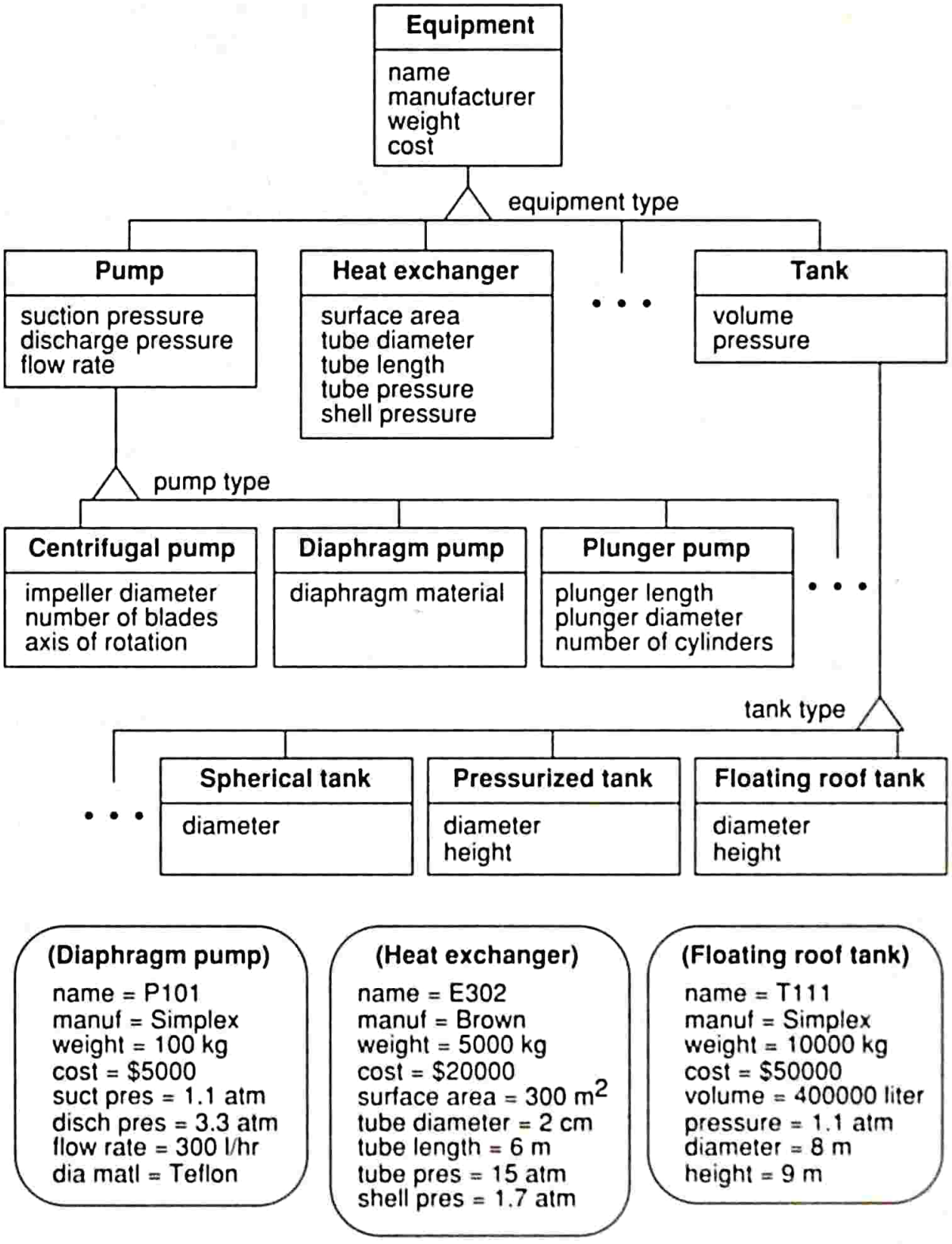


Figure 3.23 A multilevel inheritance hierarchy with instances

Boat is *water*. The discriminator is an optional part of a generalization relationship; if a discriminator is included, it should be drawn next to the generalization triangle.

Figure 3.24 shows classes of graphic geometric figures. This example has more of a programming flavor and emphasizes inheritance of operations. *Move*, *select*, *rotate*, and *display*

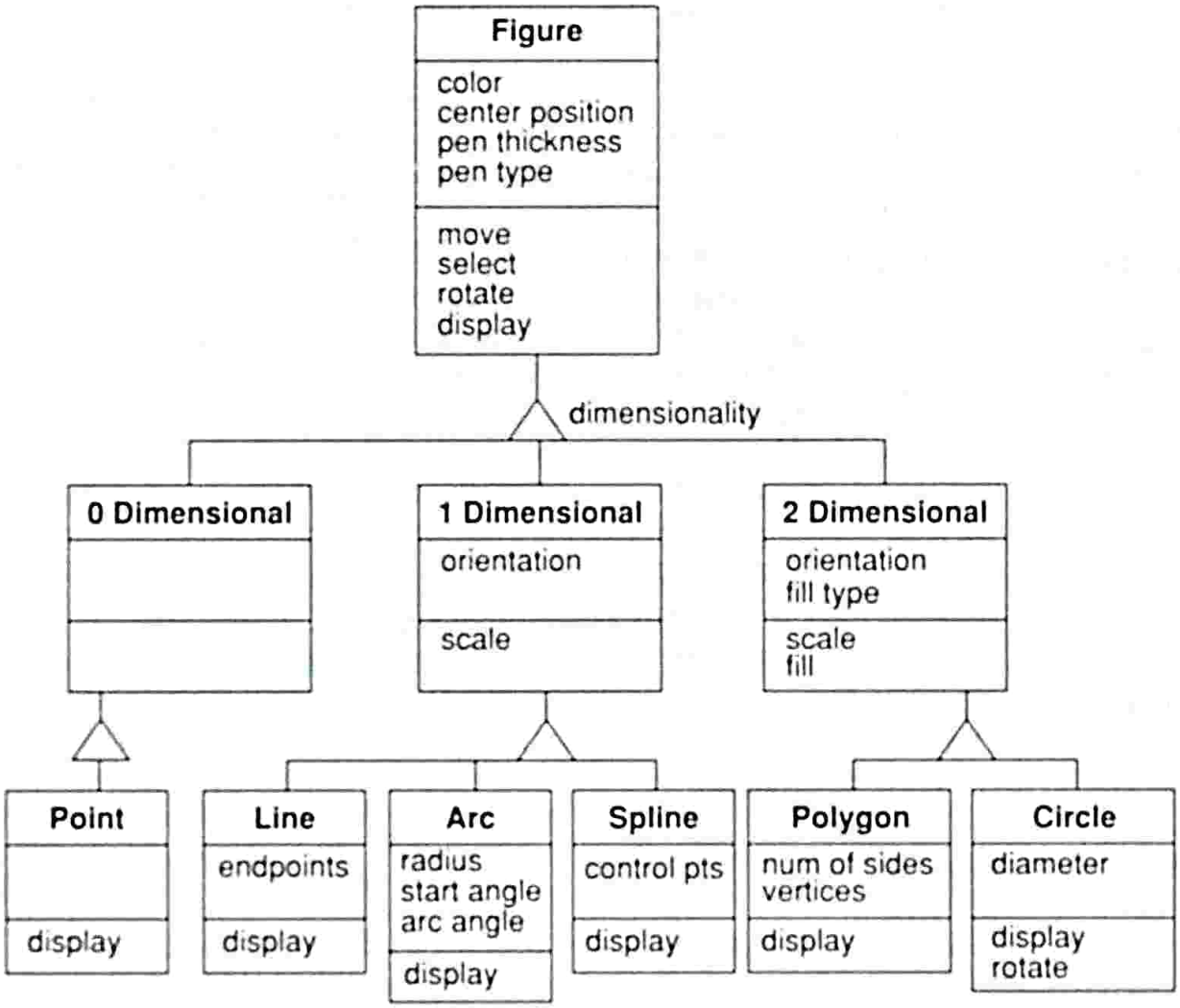


Figure 3.24 Inheritance for graphic figures

are operations inherited by all subclasses. *Scale* applies to one- and two-dimensional figures. *Fill* applies only to two-dimensional figures.

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.

3.4.2 Use of Generalization

Generalization is a useful construct for both conceptual modeling and implementation. Generalization facilitates modeling by structuring classes and succinctly capturing what is similar and what is different about classes. Inheritance of operations is helpful during implementation as a vehicle for reusing code.

Object-oriented languages provide strong support for the notion of inheritance. (The concept of inheritance was actually invented far earlier, but object-oriented languages made it popular.) In contrast, current database systems provide little or no support for inheritance. Object-oriented database programming languages (Section 15.8.5) and extended relational database systems (Section 17.4) show promise of correcting this situation.

Inheritance has become synonymous with code reuse within the object-oriented programming community. After modeling a system, the developer looks at the resulting classes and tries to group similar classes together and reuse common code. Often code is available from past work (such as a class library) which the developer can reuse and modify, where necessary, to get the precise desired behavior. The most important use of inheritance, however, is the conceptual simplification that comes from reducing the number of independent features in a system.

The terms inheritance, generalization, and specialization all refer to aspects of the same idea and are often used interchangeably. We use *generalization* to refer to the relationship among classes, while *inheritance* refers to the mechanism of sharing attributes and operations using the generalization relationship. Generalization and specialization are two different viewpoints of the same relationship, 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. In practice, there is little danger of confusion.

3.4.3 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 for better performance. For example, in Figure 3.24, *display* must be implemented separately for each kind of figure, although it is defined for any kind of figure. Operation *rotate* is overridden for performance in class *Circle* to be a null operation. Chapter 4 discusses overriding features in more detail.

You may override default values of attributes and methods of operations. 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* (Section 4.3) 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 *rotate-circle* in Figure 3.24).

A feature should never be overridden so that it is inconsistent with the signature or semantics of 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 object-oriented 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. (See Section 4.3 for further discussion of overrides.)

3.5 GROUPING CONSTRUCTS

3.5.1 Module

A *module* is a logical construct for grouping classes, associations, and generalizations. A module captures one perspective or view of a situation. For example, electrical, plumbing, and ventilation modules are different views of a building. The boundaries of a module are somewhat arbitrary and subject to judgment.

An object model consists of one or more modules. Modules enable you to partition an object model into manageable pieces. Modules provide an intermediate unit of packaging between an entire object model and the basic building blocks of class and association. Class names and association names must be unique within a module. As much as possible, you should use consistent class and association names across modules. The module name is usually listed at the top of each sheet. There is no other special notation for modules.

The same class may be referenced in different modules. In fact, referencing the same class in multiple modules is the mechanism for binding modules together. There should be fewer links between modules (external binding) than within modules (internal binding).

3.5.2 Sheet

A complex model will not fit on a single piece of paper. A *sheet* is the mechanism for breaking a large object model down into a series of pages. A sheet is a single printed page. Each module consists of one or more sheets. As a rule, we never put more than one module per sheet. A sheet is just a notational convenience, not a logical construct.

Each sheet has a title and a name or number. Each association and generalization appears on a single sheet. Classes may appear on multiple sheets. Multiple copies of the same class form the bridge for connecting sheets in an object model. Sheet numbers or sheet names inside circles contiguous to a class box indicate other sheets that refer to a class. Use of sheet cross-reference circles is optional.

3.6 A SAMPLE OBJECT MODEL

Figure 3.25 shows an object model of a workstation window management system, such as the X Window System or SunView. This model is greatly simplified—a real model of a windowing system would require a number of pages—but it illustrates many object modeling constructs and shows how they fit together into a large model.

4

Advanced Object Modeling

Chapter 4 continues our discussion of object modeling concepts with treatment of advanced topics such as aggregation, inheritance, metadata, and constraints. This chapter provides subtleties for improved modeling that can be skipped upon a first reading of this book.

4.1 AGGREGATION

Aggregation is a strong form of association in which an aggregate object is *made of* components. Components 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. A single aggregate object may have several parts; each part-whole relationship is treated as a separate aggregation in order to emphasize the similarity to association. Parts may or may not exist apart from the aggregate or appear in multiple aggregates. Aggregation is inherently transitive; an aggregate has parts, which may in turn have parts. Many aggregate operations imply transitive closure* and operate on both direct and indirect parts. Recursive aggregation is common.

* 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 which are directly connected, nodes connected by two edges, nodes connected by three edges, and so forth.

4.1.1 Aggregation Versus Association

Aggregation is a special form of association, not an independent concept. Aggregation adds semantic connotations in certain cases. 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*?
- Are some operations on the whole automatically applied to its parts?
- Are some attribute values propagated from the whole to all or some parts?
- Is there an intrinsic asymmetry to the association, where one object class is subordinate to the other?

Aggregations include part explosions and expansions of an object into constituent parts. In Figure 4.1 a company is an aggregation of its divisions, which are in turn aggregations of their departments; a company is indirectly an aggregation of departments. A company is not an aggregation of its employees, since company and person are independent objects of equal stature.

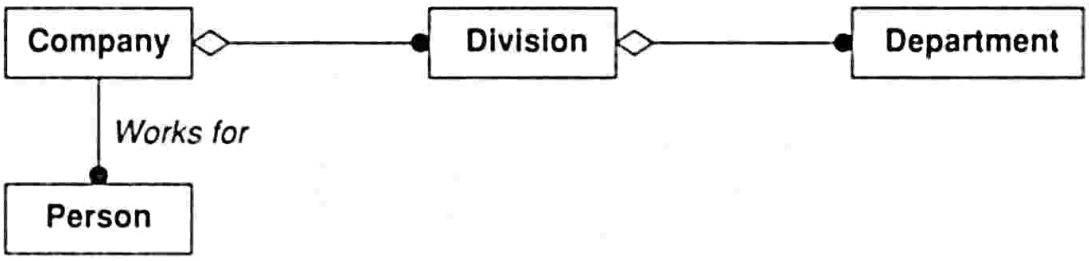


Figure 4.1 Aggregation and association

The decision to use aggregation is a matter of judgment and is often 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.

4.1.2 Aggregation Versus Generalization

Aggregation is not the same thing as generalization. Aggregation relates instances. Two distinct objects are involved; one of them is a part of the other. Generalization relates classes and is a way of structuring the description of a single object. Both superclass and subclass refer to properties of a single object. With generalization, an object is simultaneously an instance of the superclass and an instance of the subclass. Confusion arises because both aggregation and generalization give rise to trees through transitive closure. An aggregation tree is composed of object instances that are all part of a composite object; a generalization tree

is composed of classes that describe an object. Aggregation is often called "a-part-of" relationship; generalization is often called "a-kind-of" or "is-a" relationship.

Figure 4.2 illustrates aggregation and generalization for the case of a desk lamp. Parts explosions are the most compelling examples of aggregation. Base, cover, switch, and wiring are all part of a lamp. Lamps may be classified into several different subclasses: fluorescent and incandescent, for example. Each subclass may have its own distinct parts. For example, a fluorescent lamp has a ballast, twist mount, and starter; an incandescent lamp has a socket.

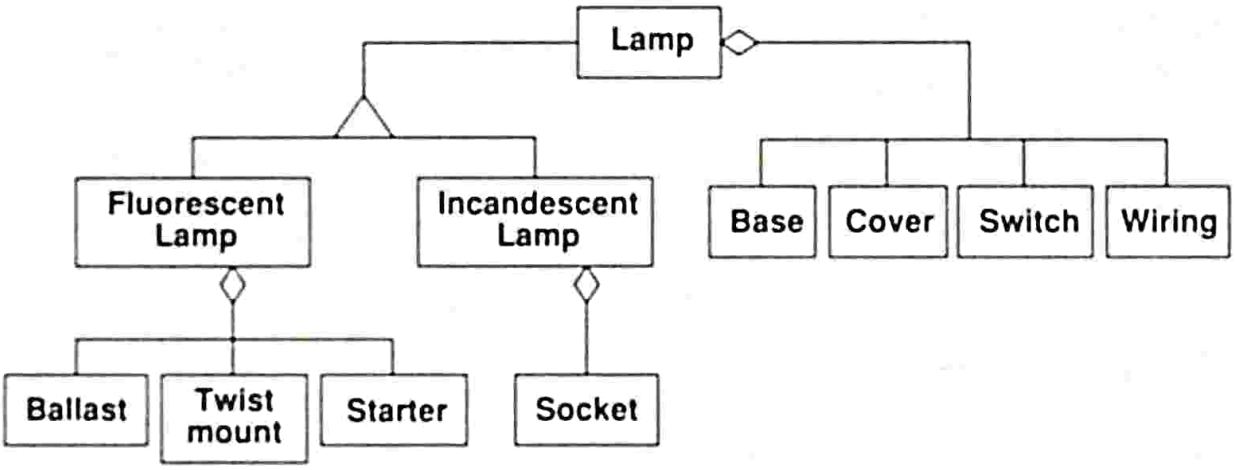


Figure 4.2 Aggregation and generalization

Aggregation is sometimes called an "and-relationship" and generalization an "or-relationship." A lamp is made of a base and a cover and a switch and wiring and so on. A lamp is a fluorescent lamp or an incandescent lamp.

4.1.3 Recursive Aggregates

Aggregation can be fixed, variable, or recursive. A fixed aggregate has a fixed structure; the number and types of subparts are predefined. The lamp in Figure 4.2 has a fixed aggregate structure.

A variable aggregate has a finite number of levels, but the number of parts may vary. The company in Figure 4.1 is a variable aggregate with a two-level tree structure. There are many divisions per company and many departments per division.

A recursive aggregate contains, directly or indirectly, an instance of the same kind of aggregate; the number of potential levels is unlimited. Figure 4.3 shows the example of a computer program. A computer program is an aggregation of blocks, with optionally recursive compound statements; the recursion terminates with simple statements. Blocks can be nested to arbitrary depth.

Figure 4.3 illustrates the usual form of a recursive aggregate: a superclass and two subclasses, one of which is an intermediate node of the aggregate and one of which is a terminal node of the aggregate. The intermediate node is an assembly of instances of the abstract superclass.

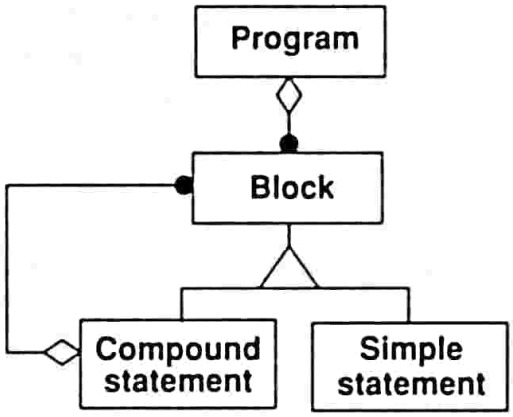


Figure 4.3 Recursive aggregate

4.1.4 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 4.4 shows an example of propagation. A person owns multiple documents. Each document is composed of paragraphs that are, in turn, composed 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.

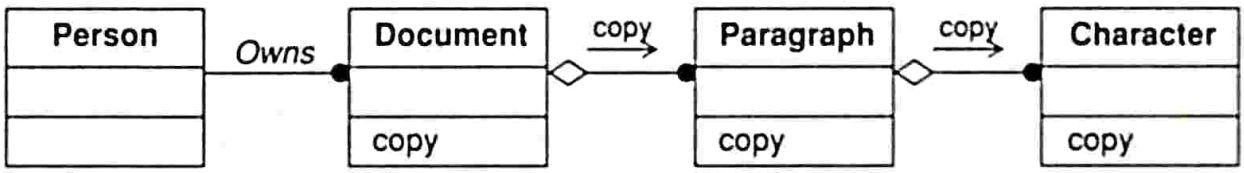


Figure 4.4 Propagation of operations

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 an entire continuum of behavior. An operation can be thought of 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.

Propagation is indicated on object models with a special notation. The propagation behavior is bound to an association (or aggregation), direction, and operation. Propagation is

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

indicated with a small arrow and operation name next to the affected association. The arrow indicates the direction of propagation.

4.2 ABSTRACT CLASSES

An *abstract class* is a class that has no direct instances but whose descendent 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). A concrete class may be a leaf class in the inheritance tree; only concrete classes may be leaf classes in the inheritance tree. Figure 4.5 summarizes the definition of abstract and concrete class. (The dotted line is the object modeling notation for instantiation and is discussed in Section 4.5.1.)

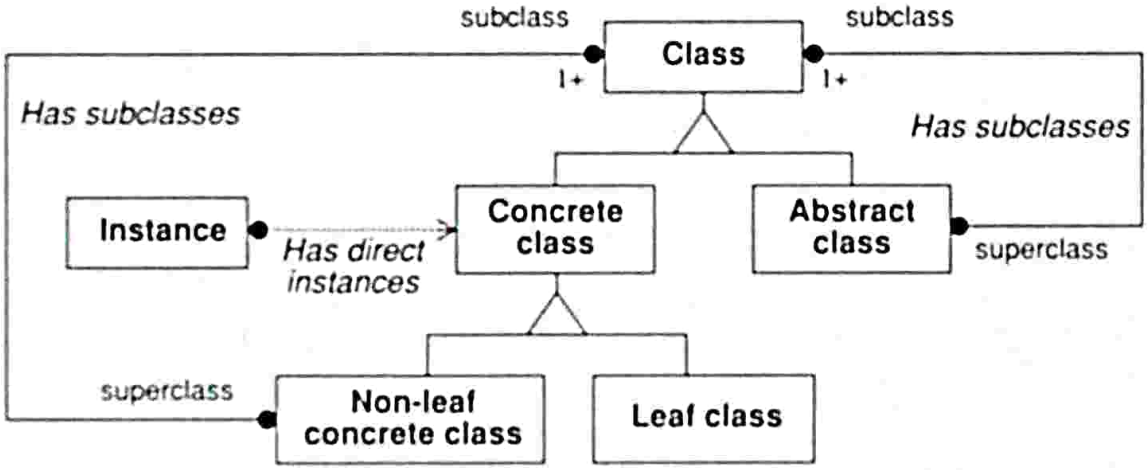


Figure 4.5 Object model defining abstract and concrete class

All the occupations shown in Figure 4.6 are concrete classes. *Butcher, Baker,* and *Candlestick Maker* are concrete classes because they have direct instances. The ellipsis notation (...) indicates that additional subclasses exist but have been omitted from the diagram, perhaps for lack of space or lack of relevance to the present concern. *Worker* also is a concrete class because some occupations may not be further specified.

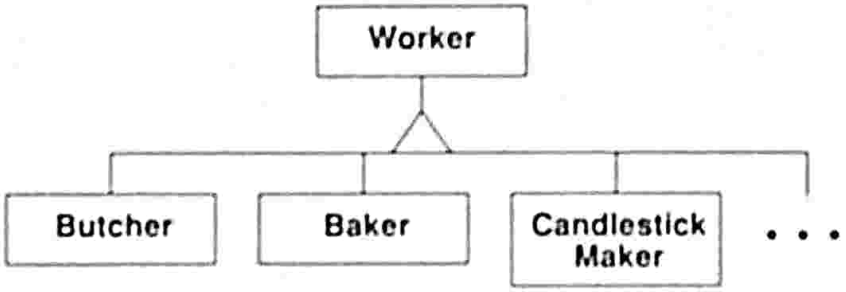


Figure 4.6 Concrete classes

Class *Employee* in Figure 4.7 is an example of an abstract class. All employees must be either hourly, salaried, or exempt.

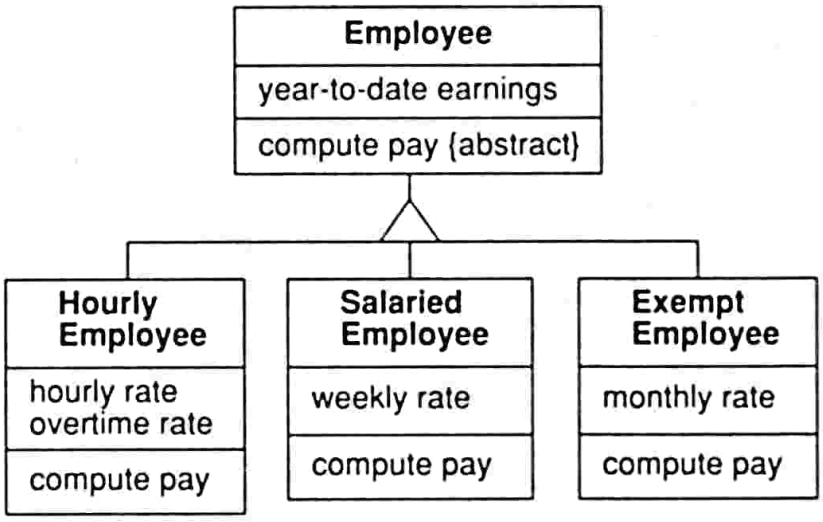


Figure 4.7 Abstract class and abstract operation

Abstract classes organize features common to several classes. It is often useful to create an abstract superclass to encapsulate classes that participate in the same association or aggregation. Some abstract classes appear naturally in the application domain. Other abstract classes are artificially introduced as a mechanism for promoting code reuse.

Abstract classes are frequently used to define methods to be inherited by subclasses. On the other hand, an abstract class can define the protocol for an operation without supplying a corresponding method. We call this an abstract operation. (Recall that in Chapter 3 we defined an operation as the protocol for an action that may be applied to objects in a class. A method is the actual implementation of an operation.) An abstract operation defines the form 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 4.7 shows an abstract operation. An abstract operation is designated by a comment in braces. *Compute-pay* is an abstract operation of class *Employee*; its form but not its implementation is defined. Each subclass must supply a method for this operation.

The *origin class* of a feature is the topmost defining class. The origin class defines the *protocol* of the feature, that is the type of an attribute or the number and type of arguments and result type for operations, as well as the semantic intent. Descendent classes can refine the protocol by further restricting the types or by overriding the initialization or method code. Descendent classes may not expand or change the protocol.

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

4.3 GENERALIZATION AS EXTENSION AND RESTRICTION

An instance of a class is an instance of all ancestors of the class. This is part of the definition of generalization. Therefore all ancestor class features must apply to the subclass instances. A descendent class cannot omit or suppress an ancestor attribute because then it would not truly be an ancestor instance. Similarly operations on an ancestor class must apply to all descendent classes. A subclass may reimplement an operation for reasons of efficiency but cannot change the external protocol.

A subclass may add new features. This is called *extension*. For example, Figure 4.7 extends class *Employee* with three subclasses that inherit all *Employee* features and add new features of their own.

A subclass may also constrain ancestor attributes. This is called *restriction* because it restricts the values that instances can assume. For example, a circle is an ellipse whose major and minor axes are equal. Arbitrary changes to the attribute values of a restricted subclass may cause it to violate the constraints, such that the result no longer belongs to the original subclass. This is not a problem from the perspective of the superclass because the result is still a valid superclass instance. For example, a circle that is scaled unequally in the x and y dimensions remains an ellipse but is no longer a circle. Class *Ellipse* is closed under the scaling operation, but *Circle* is not.

Inherited features can be renamed in a restriction. The inherited major and minor axes of a circle must be equal and could be renamed the *diameter*.

Class membership can be defined in two ways: implicitly by rule or explicitly by enumeration. A rule defines a condition for membership in a class; all objects whose values satisfy the rule belong to the class. This is the usual mathematical usage. Polygons, triangles, ellipses, circles, and other mathematical objects are defined by rule. This works well for immutable values but not so well for objects that can undergo changes yet remain the same object. Most object-oriented languages consider an object to be a discrete unit with explicit properties, one of which is the class of the object. An object has explicit class membership and the attributes it bears flows from its class. In contrast, for rule-based definition, class membership flows from attribute values.

In an explicit definition of class membership, operations that would invalidate class membership constraints must be disallowed on semantic grounds. Restriction implies that a subclass may not inherit all the operations of its ancestors. In an ideal world, such operations would be automatically detected by a support system, but for now they must be specified by the designer. Thus the *Circle* class must suppress the unequal scale operation. On the other hand, an object declared to be an ellipse is not restricted to remain a circle even if its major and minor axes happen to be temporarily equal.

Failing to note the difference between restriction and extension has caused confusion in the past. Some authors have been bothered by the fact that subclasses must suppress some operations. Chapter 10 of [Meyer-88] notes that subclasses can be viewed as both specializing and extending superclass features. These meanings are complementary. Some operations are meaningful only to a subset of instances; narrowing the set of instances broadens the number of applicable operations. Meyer also notes that the internal implementation of an operation can be overridden, provided the external protocol remains the same.

4.3.1 Overriding Operations

There is tension between use of inheritance for abstract data types and for sharing implementation. Most of this tension relates to overriding methods. The trouble arises when the overriding method substantially differs from the overridden method, rather than just refining it. Overriding is done for the following reasons:

Overriding for extension. The new operation is the same as the inherited operation, except it adds some behavior, usually affecting new attributes of the subclass. This concept is supported by Eiffel (redefine) and Smalltalk (SUPER). For example, *Window* may have a *draw* operation that draws the window boundary and contents. *Window* could have a subclass called *LabeledWindow* that also has a *draw* operation. The *draw-LabeledWindow* method could be implemented by invoking the method to draw a *Window* and then adding code to draw the label.

Overriding for restriction. The new operation restricts the protocol, such as tightening the types of arguments. This may be necessary to keep the inherited operation closed within the subclass. For example, the superclass *Set* may have the operation *add(object)*. The subclass *IntegerSet* would then have the more restrictive operation *add(integer)*.

Overriding for optimization. An implementation can take advantage of the constraints imposed by a restriction to improve the code for an operation, and this is a valid use of overriding. The new method must have the same external protocol and results as the old one, but its internal representation and algorithm may differ completely.

For example, superclass *IntegerSet* could have an operation to find the maximum integer. The method for finding the maximum of an *IntegerSet* may be implemented as a sequential search. The subclass *SortedIntegerSet* could provide a more efficient implementation of the *maximum* operation, since the contents of the set are already sorted.

Overriding for convenience. A common practice in developing new classes is to look for a class similar to what is desired. The new class is made a subclass of the existing class and overrides the methods that are inconvenient. This ad hoc use of inheritance is semantically wrong and leads to maintenance problems because there is no inherent relationship between the parent and child classes. A better approach is to generalize the common aspects of the original and new classes into a third class, from which the first two classes both inherit.

We propose the following semantic rules for inheritance. Adherence to these rules will make your software easier to understand, easier to extend, and less prone to errors of oversight.

- All query operations (operations that read, but do not change, attribute values) are inherited by all subclasses.
- All update operations (operations that change attribute values) are inherited across all extensions.
- Update operations that change constrained attributes or associations are blocked across a restriction. For example, the *scale-x* operation is permitted for class *Ellipse*, but must be blocked for subclass *Circle*.
- Operations may not be overridden to make them behave differently (in their externally-visible manifestations) from inherited operations. All methods that implement an operation must have the same protocol.

- Inherited operations can be refined by adding additional behavior.

The implementation and use of many existing object-oriented languages violates these principles.

4.4 MULTIPLE INHERITANCE

Multiple inheritance permits a class to have more than one superclass and to inherit features from all parents. This permits mixing of 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. It brings object modeling closer to the way people think. The disadvantage is a loss of conceptual and implementation simplicity. In principle, all kinds of different mixing rules can be defined to resolve conflicts among features defined on different paths.

4.4.1 Definition

A class may inherit features from more than one superclass. A class with more than one superclass is called a *join class*. A feature from the same ancestor class found along more than one path is inherited only once; it is the same feature. Conflicts among parallel definitions create ambiguities that must be resolved in implementations. In practice, such conflicts should be avoided or explicitly resolved to avoid ambiguities or misunderstandings, even if a particular language provides a priority rule for resolving conflicts.

In Figure 4.8, *AmphibiousVehicle* is both *LandVehicle* and *WaterVehicle*. In Figure 4.9, *VestedHourlyEmployee* is both *VestedEmployee* and *HourlyEmployee*. *AmphibiousVehicle* and *VestedHourlyEmployee* are join classes.

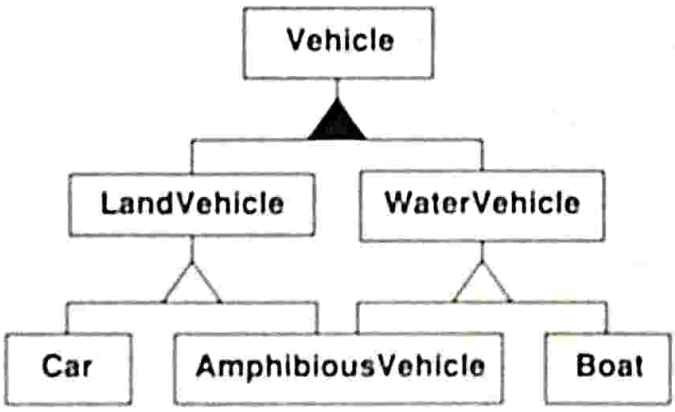


Figure 4.8 Multiple inheritance from overlapping classes

Each generalization should cover a single property, for example where a vehicle travels. If a class can be refined on several distinct and independent dimensions, then use multiple generalizations. Recall that the content of an object model is driven by its relevance to an

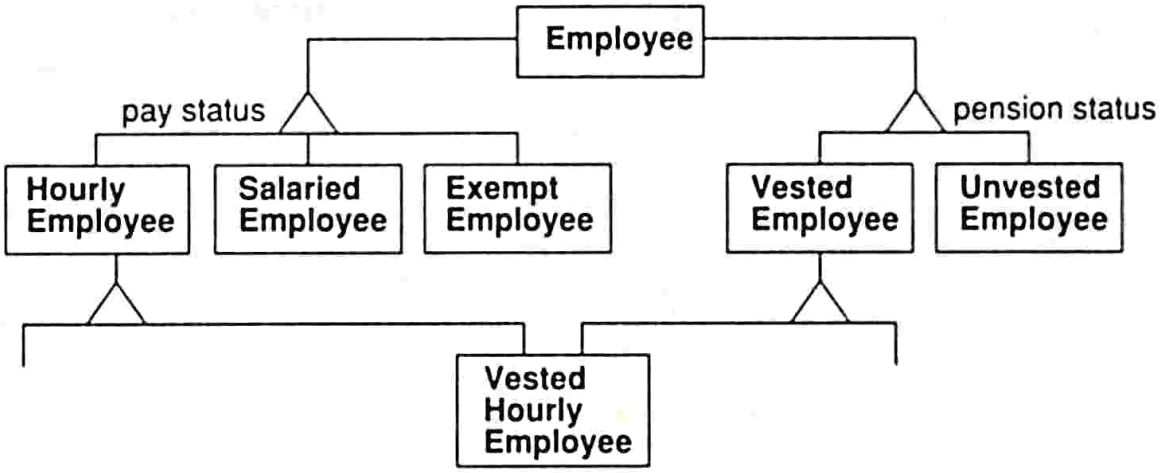


Figure 4.9 Multiple inheritance from disjoint classes

application solution, so do not list all possible generalizations, just show the important ones. In Figure 4.9, class *Employee* independently specializes on pay status and pension status. This is shown with two separate generalizations.

The generalization subclasses may or may not be disjoint. For example, *LandVehicle* and *WaterVehicle* overlap because some vehicles travel on both land and water. *HourlyEmployee*, *SalariedEmployee*, and *ExemptEmployee* are disjoint; each employee must belong to exactly one of these. A hollow triangle indicates disjoint subclasses; a solid triangle indicates overlapping subclasses. A class can multiply inherit from distinct generalizations or from different classes within an overlapping generalization but never from two classes in the same disjoint generalization.

The term *multiple inheritance* is used somewhat imprecisely to mean either the conceptual relationship between classes or the language mechanism that implements that relationship by sharing of behavior and data. Whenever possible, we try to distinguish between *generalization* (the conceptual relationship) and *inheritance* (the language mechanism), but in the case of multiple inheritance the term is so widely used already that use of the term "multiple generalization" would be confusing.

4.4.2 Accidental Multiple Inheritance

An instance of a join class is inherently an instance of all the ancestors of the join class. For example, an instructor is inherently 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 "accidental" multiple inheritance, in which one instance happens to participate in two overlapping classes. This case is poorly handled by most object-oriented languages. As shown in Figure 4.10, 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 to the OO framework [McAllester-86].

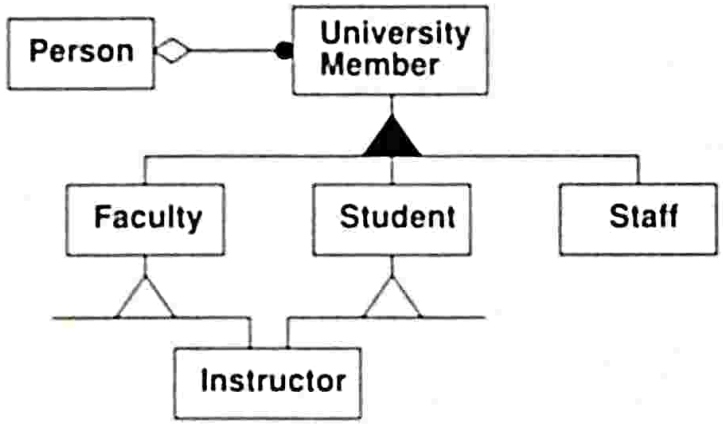


Figure 4.10 Workaround for accidental multiple inheritance

4.4.3 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. Some restructuring techniques are described below. Two of the following approaches make use of *delegation*, which is an implementation mechanism by which an object forwards an operation to another object for execution. See Section 10.6.3 for further discussion of delegation.

Delegation using aggregation of roles. A superclass with multiple independent generalizations can be recast as an aggregate in which each component replaces a generalization. This approach is similar to that for accidental multiple inheritance 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 aggregation is not automatic. They must be caught by the join class and delegated to the appropriate component.

For example, in Figure 4.11 *EmployeePayroll* becomes a superclass of *HourlyEmployee*, *SalariedEmployee*, and *ExemptEmployee*. *EmployeePension* becomes a superclass of *VestedEmployee* and *UnvestedEmployee*. Then *Employee* can be modeled as an aggregation of *EmployeePayroll* and *EmployeePension*. An operation such as *compute-pay* sent to an *Employee* object would have to be redirected to the *EmployeePayroll* component by the *Employee* class.

In this approach, the various join classes need not actually be created as explicit classes. All combinations of subclasses from the different generalizations are possible.

Inherit the most important class and delegate the rest. Figure 4.12 makes a join class a subclass of its most important superclass. The join class is treated as an aggregation of the remaining superclasses and their operations are delegated as in the previous alternative. This approach preserves identity and inheritance across one generalization.

Nested generalization. Factor on one generalization first, then the other. This approach multiplies out all possible combinations. For example, in Figure 4.13 under each of *HourlyEmployee*, *SalariedEmployee*, and *ExemptEmployee*, add two subclasses for vested and unvested employees. This preserves inheritance but duplicates declarations and code and violates the spirit of object-oriented programming.

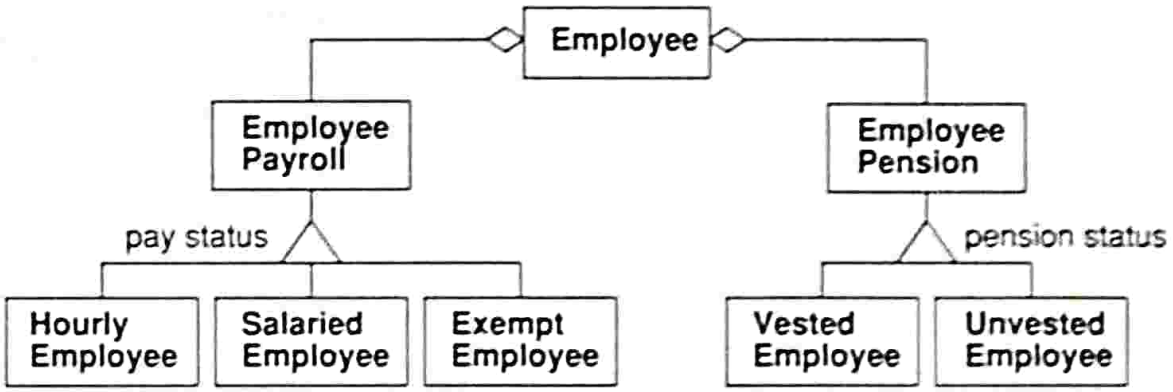


Figure 4.11 Multiple inheritance using delegation

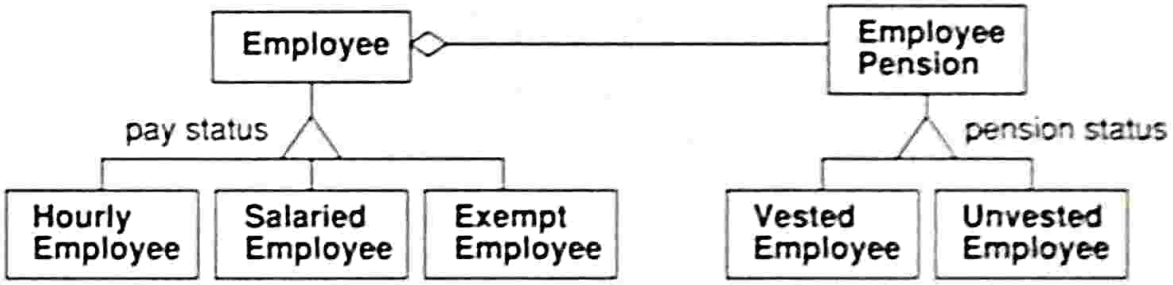


Figure 4.12 Multiple inheritance using inheritance and delegation

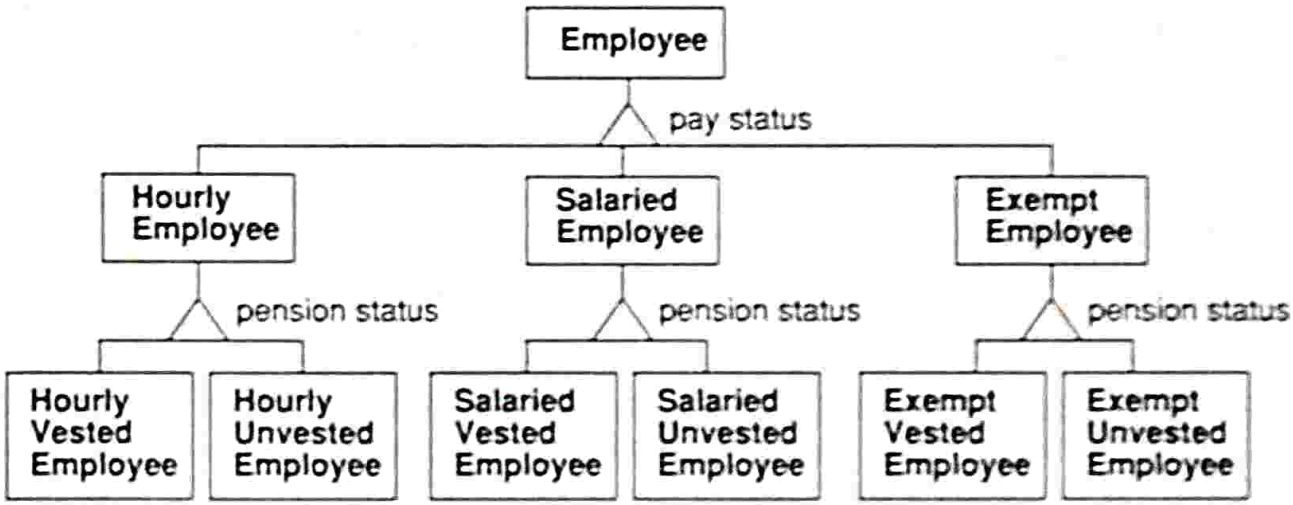


Figure 4.13 Multiple inheritance using nested generalization

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

- If a subclass has several superclasses, all of equal importance, it may be best to use delegation (Figure 4.11) and preserve symmetry in the model.

- If one superclass clearly dominates and the others are less important, implementing multiple inheritance via single inheritance and delegation may be best (Figure 4.12).
- If the number of combinations is small, consider nested generalization (Figure 4.13). If the number of combinations is large, avoid it.
- If one superclass has significantly more features than the other superclasses or one superclass clearly is the performance bottleneck, preserve inheritance through this path (Figure 4.12 or Figure 4.13).
- If you choose to use nested generalization (Figure 4.13), factor on the most important criterion first, the next most important second, and so forth.
- Try to avoid nested generalization (Figure 4.13) if large quantities of code must be duplicated.
- Consider the importance of maintaining strict identity. Only nested generalization (Figure 4.13) preserves this.

4.5 METADATA

Metadata is data that describes other data. For example, the definition of a class 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 4.5 is another example of metadata. In Section 4.2 while explaining the modeling concept of concrete and abstract classes we found it useful to use an object model to explain object modeling constructs. The case study in Chapter 18 presents an actual application that required a model of metadata (a metamodel).

Relational database management systems (see Chapter 17) also use metadata. A person can define database tables for storing information. Similarly, a relational DBMS has several metatables that store table definitions. Thus a data table may store the fact that the capital of Japan is Tokyo, the capital of Thailand is Bangkok, and the capital of India is New Delhi. A metatable would store the fact that a country has a capital city.

Metadata is frequently confusing because it blurs the normal separation between the model and the real world. With ordinary applications, the same terms can be used to refer to both the model and the real world; the context of usage distinguishes which is meant. With metadata, the context is not sufficient to distinguish the description from the thing being described, so a more precise distinction must be made.

4.5.1 Patterns and Metadata

A class describes a set of object instances of a given form. *Instantiation* relates a class to its instances. In a broader sense, any pattern describes examples of the pattern; the relationship between pattern and example can be regarded as an extension of instantiation.

Figure 4.14 shows an example of instantiation. *Joe Smith* and *Mary Wilson* are instances of class *Person*. The dotted arrows connect the instances to the class. Explicitly showing the instantiation relationship is useful when both instances and classes must be manipulated as objects, for example in interpreters, modeling tools, and language support mechanisms. Instantiation is also useful for documenting examples and test cases. For most problems, however, classes and their instances need not be shown together.

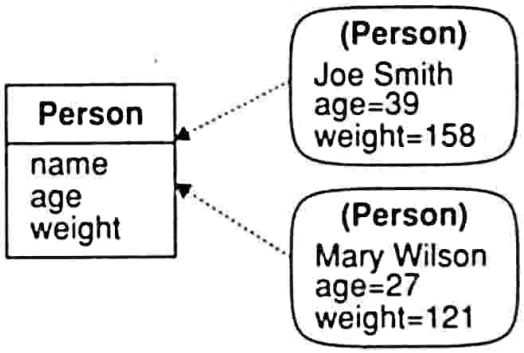


Figure 4.14 Notation for instantiation

Real-world things may be metadata. There are real-world things that describe other real-world things. A part description in a catalog describes manufactured parts. A blueprint describes a house. An engineering drawing describes a system.

For example, consider models of cars made by different manufacturers, such as a 1969 Ford Mustang or a 1975 Volkswagen Rabbit. Each *CarModel* in Figure 4.15 describes a particular kind of car; each *CarModel* has its own attributes and associations. Each *CarModel* object also describes a set of physical cars owned by persons. For example, John Doe may own a blue Ford with serial number *1FABP* and a red Volkswagen with serial number *7E81F*. Each car receives the common attributes from *CarModel* but also has its own list of particular attributes, such as serial number, color, and list of options. It would be possible to create a class to describe each kind of car, but the list of models keeps growing. It is better to consider the *CarModel* object as a pattern, a piece of metadata, that describes *Car* objects.

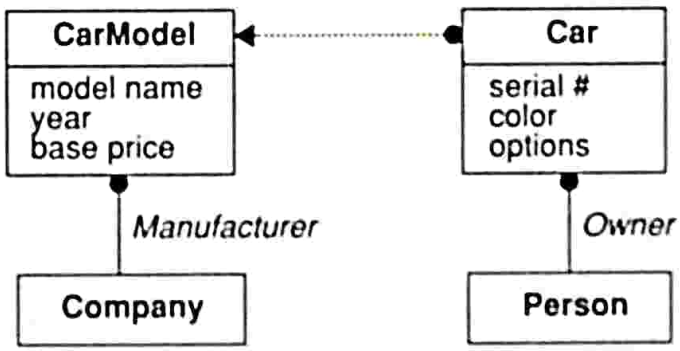


Figure 4.15 Patterns and individuals

4.5.2 Class Descriptors

Classes can also be considered as objects, but they 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.

A *class attribute* describes a value common to an entire class of objects, rather than data peculiar to each instance. Class attributes are useful for storing default information for creating new objects or summary information about instances of the class.

A *class operation* is an operation on the class itself. The most common kind of class operations are operations to create new class instances. Operations to create instances must be class operations because the instance being operated on does not initially exist. A query that provides summary information for instances in a class is also a class operation. Operations on class structure, such as scanning the list of attributes or methods, are class operations.

Figure 4.16 shows a class *Window* with class features indicated by leading dollar signs. *Window* has class attributes for the set of all windows, the set of visible windows, default window size, and maximum window size. *Window* contains class operations to create a new window object and to find the existing window with the highest priority.

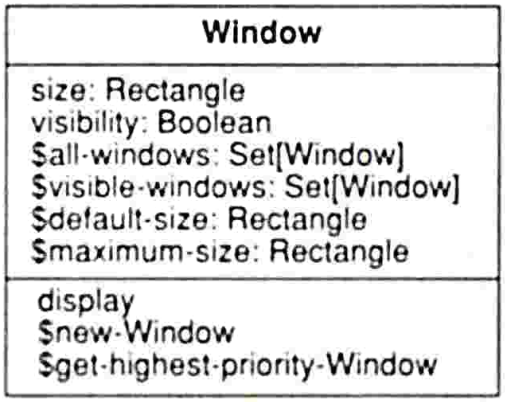


Figure 4.16 Class with class features

4.6 CANDIDATE KEYS

The ball notation discussed in Chapter 3 works fine when discussing multiplicity for binary associations and is the preferred notation for binary associations. However, multiplicity balls are ambiguous for n-ary (n>2) associations. The best approach for n-ary associations is to specify candidate keys.

A *candidate key* is a minimal set of attributes that uniquely identifies an object or link. By minimal, we mean that you cannot discard an attribute from the candidate key and still distinguish all objects and links. A class or association may have one or more candidate keys, each of which may have different combinations and numbers of attributes. The object ID is

always a candidate key for a class. One or more combinations of related objects are candidate keys for associations.

Candidate key is a term commonly used within the database community. However candidate key is really not a database concept; candidate key is a logical concept. Each candidate key constrains the instances in a class or the multiplicity of an association. Most programming languages lack the notion of a candidate key. A candidate key is delimited in an object model with braces. (This is the object modeling notation for constraints, which are discussed in the next section.)

Figure 4.17 compares multiplicity and candidate keys for binary associations. Multiplicity and candidate keys have nearly the same expressive power for binary associations. (Multiplicity also includes the notion of existence dependency—whether an object must participate in an association.) A many-to-many association requires both related objects to uniquely identify each link. A one-to-many association has a single candidate key: the object on the many end. A one-to-one association has two candidate keys: either of the objects. If we specify the country, or specify the capital city, there is no ambiguity. Note that a candidate key may be specified even when one or both classes are optional. For example, a city may not be a capital at all, but a city is capital of at most one country.

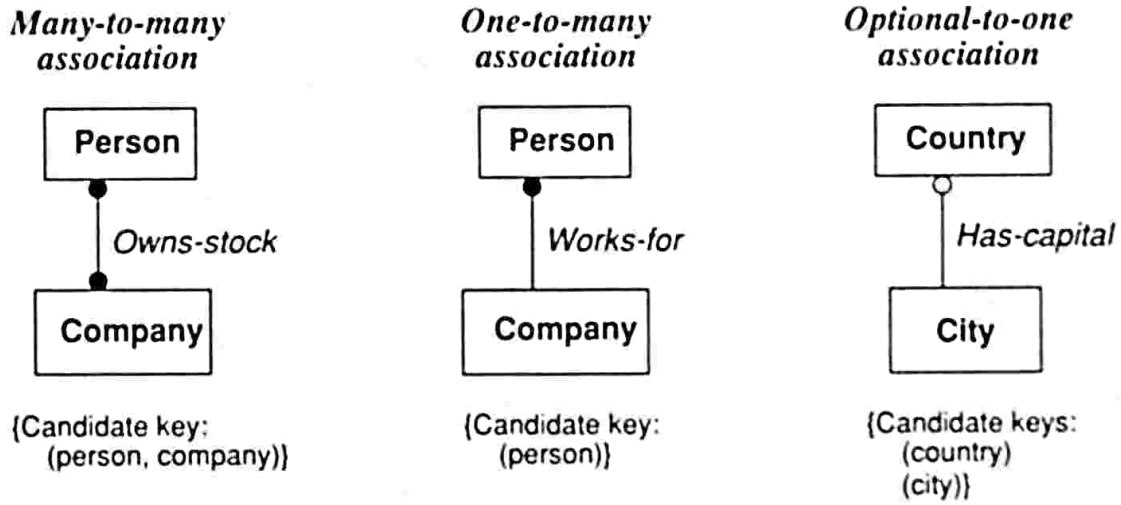
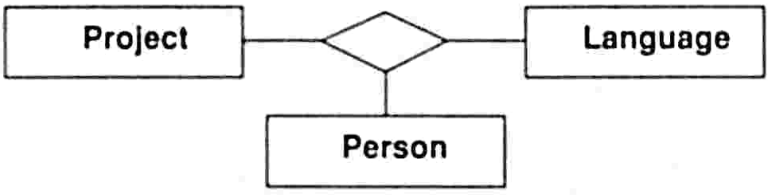


Figure 4.17 Comparison of multiplicity with candidate keys for binary associations

Figure 4.18 shows a ternary association that has one candidate key consisting of all three objects. Persons who are programmers use computer languages on projects. Several links are presented at the bottom of the figure. No combination of just one or two objects will uniquely identify each link.

Figure 4.19 contains another ternary association. A student has one advisor at a university. A student may attend more than one university. A professor may be an advisor at more than one university. Here the instances suggest that (student, university) is the only candidate key. This candidate key involves only two of the three related objects.

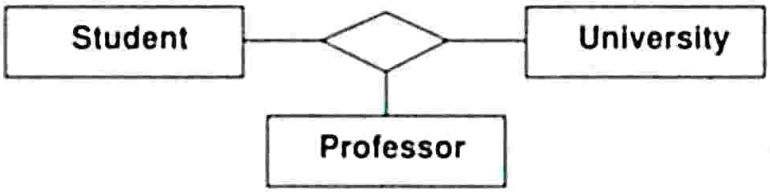
We deduce the candidate key by considering all the possibilities. *Student* is not a candidate key; two links have the value *Mary*. *Professor* and *University* are not candidate keys. *Student+Professor*, *Professor+University* are not candidate keys; *Susan+Weaver*,



{Candidate key: (project, person, language)}

Project	Person	Language
CAD program	Mary	C
control software	Susan	Ada
C++ compiler	Mike	C
CAD program	Bob	assembler
CAD program	Mike	C
CAD program	Mike	assembler

Figure 4.18 Ternary association



{Candidate key: (student, university)}

Student	Professor	University
Mary	Prof Weaver	SUNY
Mary	Prof Rumrow	RPI
Susan	Prof Weaver	RPI
Susan	Prof Weaver	SUNY
Bob	Prof Shapiro	Oxford

Figure 4.19 Ternary association

Weaver+SUNY appear twice. *Student+University* may be a candidate key, since no links share the same values. Based on the problem statement, we decide that *Student+University* really is a candidate key and would distinguish other links beyond the five in Figure 4.19. *Student+Professor+University* is not a candidate key, because it is not a minimal set of attributes.

4.7 CONSTRAINTS

4.7.1 Definition

Constraints are functional relationships between entities of an object model. The term *entity* includes objects, classes, attributes, links, and associations. A constraint restricts the values that entities can assume. Examples include: No employee's salary can exceed the salary of

the employee's boss (a constraint between two things at the same time). No window will have an aspect ratio (length/width) of less than 0.8 or greater than 1.5 (a constraint between properties of a single object). The priority of a job may not increase (constraint on the same object over time). Figure 4.20 lists these examples. Simple constraints may be placed in object models. Complex constraints should be specified in the functional model (see Chapter 6).

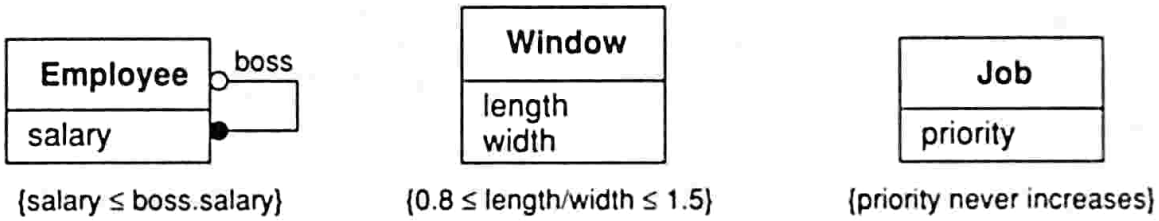


Figure 4.20 Constraints on objects

We favor expressing constraints in a declarative manner. Ordinarily, constraints must be converted to procedural form before they can be stated in a programming language. Ideally conversion should be automatic, but this may be difficult or impossible to achieve. Object models capture some constraints through their very structure. For example, single inheritance implies that subclasses are mutually exclusive.

Constraints provide one criterion for measuring the quality of an object model; a "good" object 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. In principle, we could embellish object modeling notation with all kinds of special constructs to capture more and more structural constraints. This is probably not a good idea. The object modeling notation advanced by this book represents a compromise between expressive power and simplicity. There will always be constraints that must be expressed in natural language.

Object modeling syntax for constraints is as follows: Constraints are delimited by braces and positioned near the constrained entity. A dotted line connects multiple constrained entities. An arrow may be used to connect a constrained entity to the entity it depends on. Instantiation is a kind of constraint and therefore uses the same notation.

4.7.2 Constraints on Links

Multiplicity constrains an association. It restricts the number of objects related to a given object. Object modeling notation has a special syntax for showing common multiplicity values ([0,1], exactly 1, and 0+). Other values of multiplicity can be shown by a numerical interval near an association role. For example, Figure 4.5 specifies "1+" for two association roles.

The notation "{ordered}" indicates that the elements of the "many" end of an association have an explicit order that must be preserved. Figure 4.21 shows an object model for the officers of a country. Each office (such as president, chief justice, king) for each country has a set of persons who have held the office, ordered chronologically.

Figure 4.21 Constraints on association links

4.7.3 General Constraints

General constraints must be expressed with natural language or equations. You should draw a dotted line between classes involved in the constraint and specify the details with comments in braces. Sometimes it may be impractical to draw lines to all the classes, so make the best of it. Sometimes it is better to have an unattached constraint than to draw lines all over the place.

For example, one association may be a subset of another. In Figure 4.22 the chair of a committee must be a member of the committee; the *Chair-of* association is a subset of the *Member-of* association.

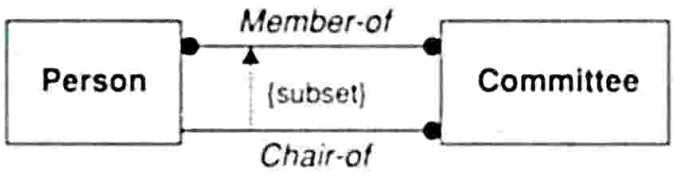


Figure 4.22 Subset constraint between associations

4.7.4 Derived Objects, Links, and Attributes

A *derived object* is defined as a function of one or more objects, which in turn may be derived. The derived object is completely determined by the other objects. Ultimately, the derivation tree terminates with base objects. Thus a derived object is redundant but may be included in an object model to ease comprehension; it often represents a meaningful real-world concept. Similarly, there are also *derived links* and *derived attributes*.

The notation for a derived entity is a slash or diagonal line (on the corner of a class box, on an association line, or in front of an attribute). You should show the constraint that determines the derived value. Like most of the object modeling notation, the derived value notation is optional.

As shown in Figure 4.23, age provides a good example of a derived attribute. Age can be derived from birth date and the current date.

In Figure 4.24, 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 object class called *Offset* related to each part by a derived association called *NetOffset*.

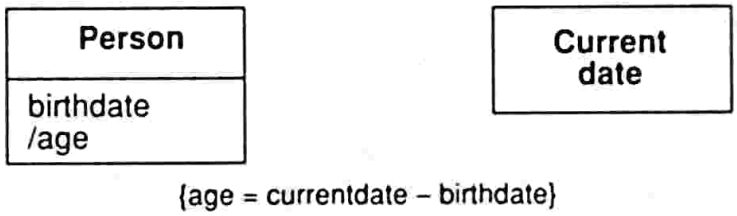


Figure 4.23 Derived attribute

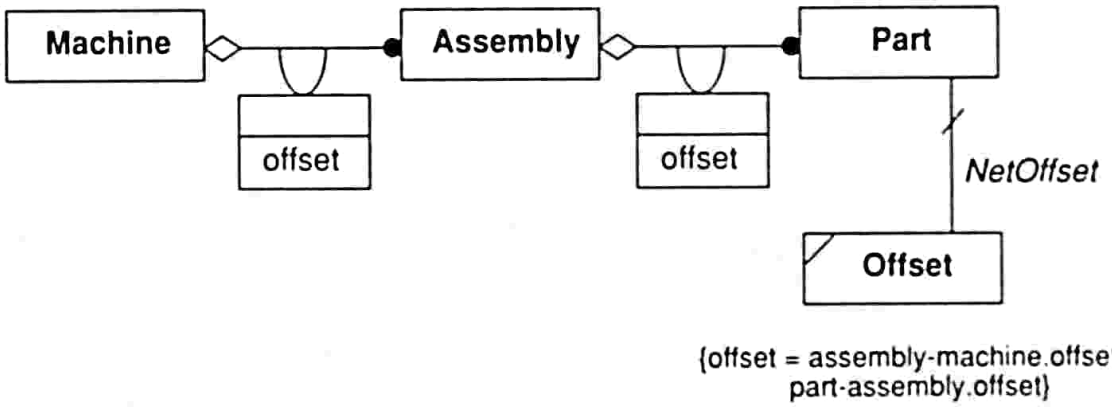


Figure 4.24 Derived object and association

Real world concepts are highly redundant, therefore we expect to see many derived entities in models; it is desirable to use concepts that appear in the application domain. Nevertheless it is important to distinguish independent and dependent entities in a model so that the true complexity can be seen. Derived entities are constrained by their base entities and the derivation rule.

4.7.5 Homomorphisms

A *homomorphism* maps between two associations as illustrated by Figure 4.25. For example, in a parts catalog for an automobile, a catalog item may contain other catalog items. Each catalog item is specified by a model number that corresponds to thousands or millions of individual manufactured items, each with its own serial number. The individual items are also composed of subitems. Each physical item's parts explosion tree has the same form as the catalog item's parts explosion tree. The *contains* aggregation on catalog items is a homomorphism of the *contains* aggregation on physical items. This form of homomorphism between two trees is common.

In general, a homomorphism involves four relationships among four classes as shown in Figure 4.26. The homomorphism maps links of one general association (u) into links of another general association (t) as a many-to-one mapping. Two instantiation relationships map elements of one class into another: r is a many-to-one mapping from class B to class A and s is a many-to-one mapping from class D to class C. In the common case where t is on a single class and u is on a single class, then A=C, B=D, and r=s, as in Figure 4.25.

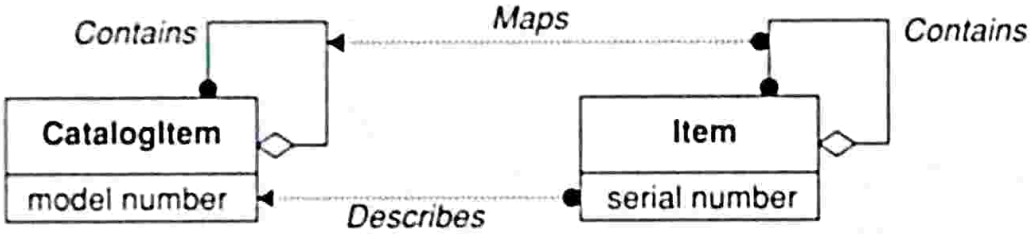


Figure 4.25 Homomorphism for a parts catalog

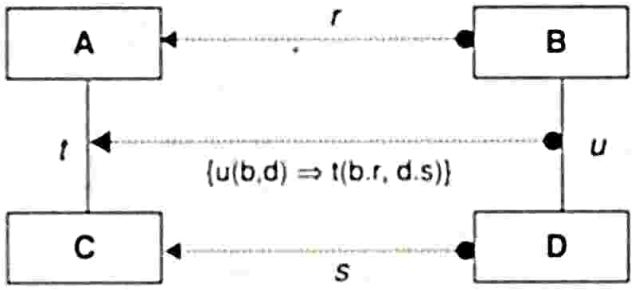


Figure 4.26 General homomorphism

At first, the homomorphism may seem to be an esoteric concept. However, our experience has been that they really do appear in practice. Homomorphisms are most likely to occur for complex applications that deal with metadata. The homomorphism is essentially just an analogy—a special type of relationship between relationships. Proper use of homomorphisms constrain the structure of an object model and improve the correspondence between the model and the real world.

4.8 CHAPTER SUMMARY

This chapter covers several diverse topics that explain subtleties of object modeling. These concepts are not needed for simple models but may be important for complex applications. Remember, the content of any object model should be driven by its relevance to an application. 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.

Aggregation is a special form of transitive association where a group of component objects form a single semantic entity. Operations on an aggregate often propagate to the components. Recursive aggregates allow a component to also be an aggregate. Aggregation must not be confused with generalization, even though both constructs form trees; aggregation is a tree of instances, generalization is a tree of classes.

Abstract and *concrete* are useful terms for referring to classes in an inheritance hierarchy. Abstract classes help organize the class hierarchy and have no direct instances. Concrete