Chapter 14

Object-Oriented Data Modeling

LEARNING OBJECTIVES

After studying this chapter, you should be able to:

- Concisely define each of the following key terms: **class, object, state, behavior, object class, class diagram, object diagram, operation, encapsulation, constructor operation, query operation, update operation, scope operation, association, association role, multiplicity, association class, abstract class, concrete class, class-scope attribute, abstract operation, method, polymorphism, overriding, multiple classification, aggregation,** and **composition.**
- Describe the activities in the different phases of the object-oriented development life cycle.
- State the advantages of object-oriented modeling vis-à-vis structured approaches.
- Compare and contrast the object-oriented model with the E-R and EER models.
- Model a real-world application by using a UML class diagram.
- Provide a snapshot of the detailed state of a system at a point in time using a UML (Unified Modeling Language) object diagram.
- Recognize when to use generalization, aggregation, and composition relationships.
- Specify different types of business rules in a class diagram.

INTRODUCTION

In Chapters 3 and 4, you learned about data modeling using the E-R and EER models. In those chapters, you learned how to model the data needs of an organization using entities, attributes, and a wide variety of relationships. In this chapter, you will be introduced to the object-oriented model, which is becoming increasingly popular because of its ability to represent thoroughly complex relationships, as well as to represent data and data processing in a consistent notation. Fortunately, most of the concepts you learned in those chapters correspond to concepts in object-oriented modeling, but, as you will see, the object-oriented model has even more features than the EER model.

As you learned in Chapters 3 and 4, a data model is an abstraction of the real world. It allows you to deal with the complexity inherent in a real-world problem

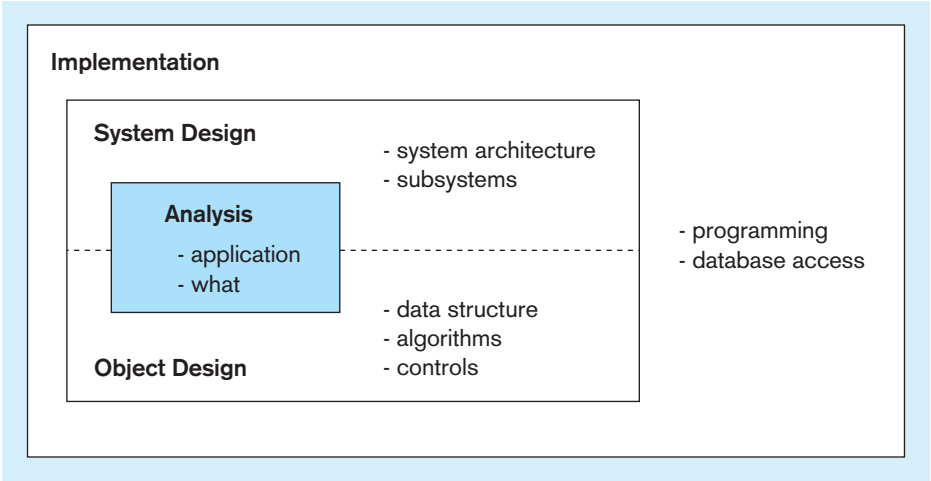


Figure 14-1
Phases of object-oriented systems development cycle

by focusing on the essential and interesting features of the data an organization needs. An object-oriented model is built around *objects,* just as the E-R model is built around entities. However, as we shall see later, an object *encapsulates* both data *and* behavior, implying that we can use the object-oriented approach not only for data modeling, but also for process modeling. To thoroughly model any real-world application, you need to model both the data and the processes that act on the data (recall the discussion in Chapter 2 about information planning objects). By allowing you to capture them together within a common representation, and by offering benefits such as *inheritance* and code reuse, the object-oriented modeling approach provides a powerful environment for developing complex systems.

The object-oriented development life cycle, depicted in Figure 14-1, consists of progressively and iteratively developing object representation through three phases—analysis, design, and implementation—similar to the heart of the systems development life cycle explained in Chapter 2. In the early stages of development, the model you develop is abstract, focusing on external qualities of the system. As the model evolves, it becomes more and more detailed, the focus shifting to how the system will be built and how it should function. The emphasis in modeling should be on analysis and design, focusing on front-end conceptual issues, rather than back-end implementation issues, which unnecessarily restrict design choices (Rumbaugh et al., 1991).

In the analysis phase, you develop a model of the real-world application, showing its important properties. The model abstracts concepts from the application domain and describes *what* the intended system must do, rather than *how* it will be done. It specifies the functional behavior of the system independent of concerns relating to the environment in which it is to be finally implemented. You need to devote sufficient time to clearly understand the requirements of the problem. The analysis model should capture those requirements completely and accurately. Remember that it is much easier and cheaper to make changes or fix flaws during analysis than during the later phases.

In the object-oriented design phase, you define how the application-oriented analysis model will be realized in the implementation environment. Jacobson et al. (1992) cite three reasons for using object-oriented design. These three reasons are:

1. The analysis model is not formal enough to be implemented directly in a programming language. To move seamlessly into the source code requires refining the objects by making decisions about what operations an object will provide, what the communication between objects should look like, what messages are to be passed, and so forth.

2. The actual system must be adapted to the environment in which the system will actually be implemented. To accomplish that, the analysis model has to be transformed into a design model, considering different factors such as performance requirements, real-time requirements and concurrency, the target hardware and systems software, the DBMS and programming language to be adopted, and so forth.
3. The analysis results can be validated using object-oriented design. At this stage, you can verify whether the results from the analysis are appropriate for building the system and make any necessary changes to the analysis model.

To develop the design model, you must identify and investigate the consequences that the implementation environment will have on the design. All strategic design decisions, such as how the DBMS is to be incorporated, how process communications and error handling are to be achieved, what component libraries are to be reused, are made. Next, you incorporate those decisions into a first-cut design model that adapts to the implementation environment. Finally, you formalize the design model to describe how the objects interact with one another for each conceivable scenario (use case).

Rumbaugh et al. (1991) separate the design activity into two stages: system design and *object design*. As the system designer, you propose an overall system architecture, which organizes the system into components called subsystems and provides the context to make decisions such as identifying concurrency, allocating subsystems to processors and tasks, handling access to global resources, selecting the implementation of control in software, and more.

During object design, you build a design model by adding implementation details (such as restructuring classes for efficiency, deciding on the internal data structures and algorithms to implement each class, implementing control, implementing associations, and packaging into physical modules) to the analysis model in accordance with the strategy established during system design. The application-domain object classes from the analysis model still remain, but they are augmented with computer-domain constructs, with a view toward optimizing important performance measures.

The design phase is followed by the implementation phase. In this phase, you implement the design using a programming language and/or a database management system. Translating the design into program code is a relatively straightforward process, given that the design model already incorporates the nuances of the programming language and the DBMS.

Coad and Yourdon (1991b) identify several motivations and benefits of object-oriented modeling, which follow:

- The ability to tackle more challenging problem domains
- Improved communication between the users, analysts, designers, and programmers
- Increased consistency among analysis, design, and programming activities
- Explicit representation of commonality among system components
- Robustness of systems
- Reusability of analysis, design, and programming results
- Increased consistency among all the models developed during object-oriented analysis, design, and programming

The last point needs further elaboration. In other modeling approaches, such as structured analysis and design (described in Chapter 2), the models that are developed (e.g., data flow diagrams during analysis and structure charts during design) lack a common underlying representation and, therefore, are very weakly connected. In contrast to the abrupt and disjoint transitions that those approaches suffer from,

the object-oriented approach provides a continuum of representation from analysis to design to implementation, engendering a seamless transition from one model to another. For instance, moving from object-oriented analysis to object-oriented design entails expanding the analysis model with implementation-related details, not developing a whole new representation.

In this chapter, we present object-oriented data modeling as a high-level conceptual activity. As you will learn in Chapter 15, a good conceptual model is invaluable for designing and implementing an object-oriented database application.

THE UNIFIED MODELING LANGUAGE

The Unified Modeling Language (UML) is "a language for specifying, visualizing, and constructing the artifacts of software systems, as well as for business modeling" (*UML Document Set*, 1997). It culminated from the efforts of three leading experts, Grady Booch, Ivar Jacobson, and James Rumbaugh, who have defined an object-oriented modeling language that is expected to become an industry standard in the near future. The UML builds upon and unifies the semantics and notations of the Booch (Booch, 1994), OOSE (Jacobson et al., 1992), and OMT (Rumbaugh et al., 1991) methods, as well as those of other leading methods. UML has recently been updated to UML2.0, maintained by the Object Management Group. We use the 1997 notation set in this chapter because it is more widely used and there are no significant changes for the purpose of this chapter.

The UML notation is useful for graphically depicting an object-oriented analysis or design model. It not only allows you to specify the requirements of a system and capture the design decisions, but it also promotes communication among key persons involved in the development effort. A developer can use an analysis or design model expressed in the UML notation as a means to communicate with domain experts, users, and other stakeholders.

For representing a complex system effectively, the model you develop must have a small set of independent views or perspectives. UML allows you to represent multiple perspectives of a system by providing different types of graphical diagrams, such as the use-case diagram, class diagram, state diagram, interaction diagram, component diagram, and deployment diagram. The underlying model integrates those views so that the system can be analyzed, designed, and implemented in a complete and consistent fashion.

Because this text is about databases, we will describe only the *class diagram,* which addresses the data, as well some behavioral, aspects of a system. We will not describe the other diagrams because they provide perspectives that are not directly related to a database system, for example, the dynamic aspects of a system. But keep in mind that a database system is usually part of an overall system, whose underlying model should encompass all the different perspectives. For a discussion of other UML diagrams, see Hoffer et al. (2005) and George et al. (2005).

OBJECT-ORIENTED DATA MODELING

In this section, we introduce you to object-oriented data modeling. We describe the main concepts and techniques involved in object-oriented modeling, including objects and classes; encapsulation of attributes and operations; association, generalization, and aggregation relationships; cardinalities and other types of constraints; polymorphism; and inheritance. We show how you can develop class diagrams, using the UML notation, to provide a conceptual view of the system being modeled.

Representing Objects and Classes

In the object-oriented approach, we model the world in objects. Before applying the approach to a real-world problem, therefore, we need to understand what an object really is. A **class** is an entity that has a well-defined role in the application domain about which the organization wishes to maintain state, behavior, and identity. A class is a concept, abstraction, or thing that makes sense in an application context (Rumbaugh et al., 1991). A class could be a tangible or visible entity (e.g., a person, place, or thing); it could be a concept or event (e.g., Department, Performance, Marriage, Registration, etc.); or it could be an artifact of the design process (e.g., User Interface, Controller, Scheduler, etc.). An **object** is an instance of a class (e.g., a particular person, place, or thing) that encapsulates the data and behavior we need to maintain about that object. A class of objects shares a common set of attributes and behaviors.

Class: An entity that has a well-defined role in the application domain about which the organization wishes to maintain state, behavior, and identity.

Object: An instance of a class that encapsulates data and behavior.

You might be wondering how classes and objects are different from entity types and entity instances in the E-R model you studied in Chapter 3. Clearly, entity types in the E-R model can be represented as classes and entity instances as objects in the object model. But, in addition to storing a state (information), an object also exhibits behavior, through operations that can examine or affect its state.

The **state** of an object encompasses its properties (attributes and relationships) and the values those properties have, and its **behavior** represents how an object acts and reacts (Booch, 1994). An object's state is determined by its attribute values and links to other objects. An object's behavior depends on its state and the operation being performed. An operation is simply an action that one object performs upon another in order to get a response. You can think of an operation as a service provided by an object (supplier) to its clients. A client sends a message to a supplier, which delivers the desired service by executing the corresponding operation.

State: Encompasses an object's properties (attributes and relationships) and the values those properties have.

Behavior: Represents how an object acts and reacts.

Consider an example of the student class and a particular object in this class, Mary Jones. The state of this object is characterized by its attributes, say, name, date of birth, year, address, and phone, and the values these attributes currently have. For example, name is "Mary Jones," year is "junior," and so on. Its behavior is expressed through operations such as calc-gpa, which is used to calculate a student's current grade point average. The Mary Jones object, therefore, packages both its state and its behavior together.

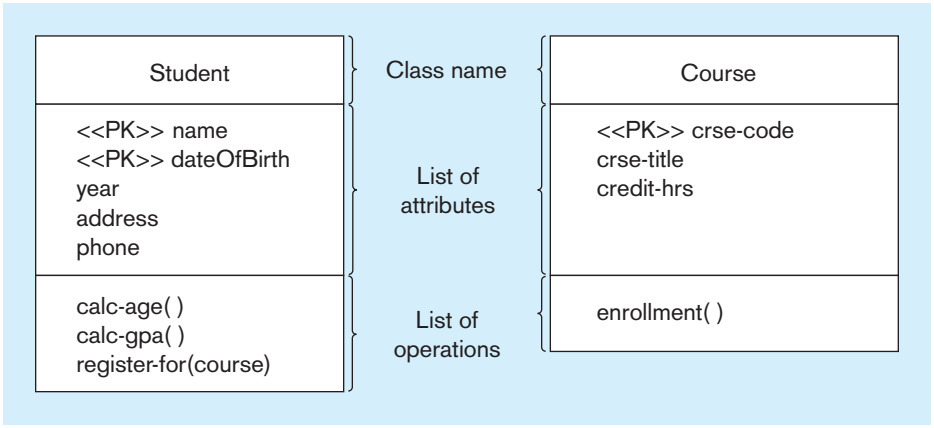
All objects have a persistent identity; that is, no two objects are the same. For example, if there are two Student instances with the same name and date of birth (these attributes have been selected as a composite primary key <<PK>> of the Student class), they are essentially two different objects. Even if those two instances have identical values for all the primary key attributes of the object, the objects maintain their separate identities. At the same time, an object maintains its own identity over its life. For example, if Mary Jones gets married and changes her name, address, and phone, she will still be represented by the same object.

You can depict the classes graphically in a class diagram as in Figure 14-2a. A **class diagram** shows the static structure of an object-oriented model: the object classes, their internal structure, and the relationships in which they participate. In UML, a class is represented by a rectangle with three compartments separated by horizontal lines. The class name appears in the top compartment, the list of attributes in the middle compartment, and the list of operations in the bottom compartment of a box. The figure shows two classes, Student and Course, along with their attributes and operations.

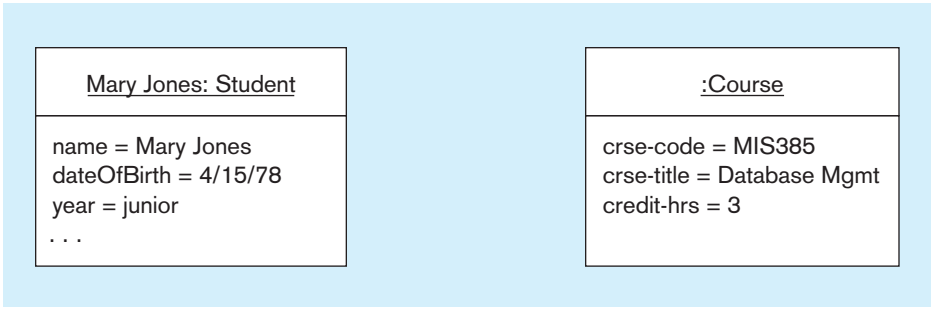
Class diagram: Shows the static structure of an object-oriented model: the object classes, their internal structure, and the relationships in which they participate.

The Student class is a group of Student objects that share a common structure and a common behavior. All students have in common the properties of name, date of birth, year, address, and phone. They also exhibit common behavior by sharing the calc-age, calc-gpa, and register-for(course) operations. A class, therefore, provides a template or schema for its instances. Each object knows to which class it

Figure 14-2
UML class and object diagrams
(a) Class diagram showing two classes



(b) Object diagram with two instances



belongs; for example, the Mary Jones object knows that it belongs to the Student class. Objects belonging to the same class may also participate in similar relationships with other objects; for example, all students register for courses and, therefore, the Student class can participate in a relationship called 'registers-for' with another class called Course (see the later section on Association).

Object diagram: A graph of instances that are compatible with a given class diagram.

An **object diagram**, also known as instance diagram, is a graph of instances that are compatible with a given class diagram. In Figure 14-2b, we have shown an object diagram with two instances, one for each of the two classes that appear in Figure 14-2a. A static object diagram, such as the one shown in the figure, is an instance of a class diagram, providing a snapshot of the detailed state of a system at a point in time.

In an object diagram, an object is represented as a rectangle with two compartments. The names of the object and its class are underlined and shown in the top compartment using the following syntax:

objectname:classname

The object's attributes and their values are shown in the second compartment. For example, we have an object called Mary Jones that belongs to the Student class. The values of the name, dateOfBirth, and year attributes are also shown. Attributes whose values are not of interest to you may be suppressed; for example, we have not shown the address and phone attributes for Mary Jones. If none of the attributes are of interest, the entire second compartment may be suppressed. The name of the object may also be omitted, in which case the colon should be kept with the class name as we have done with the instance of Course. If the name of the object is shown, the class name, together with the colon, may be suppressed.

The object model permits multivalued, composite, derived, and other types of attributes. The typical notation is to preface the attribute name with a stereotype symbol, similar to <<PK>> for primary key, that indicates its property (e.g., <<Multivalued>> for a multivalued attribute). For composite attributes, the composite

is defined as a separate class and then any attribute with that composite structure is defined as a data type of the composite class. For example, we could define a class of Address, just as we define the Student class, which is composed of Street, City, State, and Zip attributes. Then, in Figure 14-2a, if the address attribute were such a composite attribute, we would replace the address attribute line in the Student class with, for example,

stuAddress : Address

which indicates that the stuAddress attribute is of type Address. This is a powerful feature of the object model, in which we can reuse previously defined structures.

An **operation**, such as calc-gpa in Student (see Figure 14-2a), is a function or a service that is provided by all the instances of a class. It is only through such operations that other objects can access or manipulate the information stored in an object. The operations, therefore, provide an external interface to a class; the interface presents the outside view of the class without showing its internal structure or how its operations are implemented. This technique of hiding the internal implementation details of an object from its external view is known as **encapsulation**, or information hiding. So although we provide the abstraction of the behavior common to all instances of a class in its interface, we encapsulate within the class its structure and the secrets of the desired behavior.

Operation: A function or a service that is provided by all the instances of a class.

Encapsulation: The technique of hiding the internal implementation details of an object from its external view.

Types of Operations

Operations can be classified into four types, depending on the kind of service requested by clients: (1) constructor, (2) query, (3) update, and (4) scope (*UML Notation Guide*, 1997). A **constructor operation** creates a new instance of a class. For example, you can have an operation called create-student within Student that creates a new student and initializes its state. Such constructor operations are available to all classes and are therefore not explicitly shown in the class diagram.

Constructor operation: An operation that creates a new instance of a class.

A **query operation** is an operation without any side effects; it accesses the state of an object but does not alter the state (Fowler, 2000). For example, the Student class can have an operation called get-year (not shown), which simply retrieves the year (freshman, sophomore, junior, or senior) of the Student object specified in the query. Note that there is no need to explicitly show a query such as get-year in the class diagram because it retrieves the value of an independent base attribute. Consider, however, the calc-age operation within Student. This is also a query operation because it does not have any side effects. Note that the only argument for this query is the target Student object. Such a query can be represented as a derived attribute (Rumbaugh et al., 1991); for example, we can represent "age" as a derived attribute of Student. Because the target object is always an implicit argument of an operation, there is no need to show it explicitly in the operation declaration.

Query operation: An operation that accesses the state of an object but does not alter the state.

An **update operation** has side effects; it alters the state of an object. For example, consider an operation of Student called promote-student (not shown). The operation promotes a student to a new year, say, from junior to senior, thereby changing the Student object's state (value of the year attribute). Another example of an update operation is register-for(course), which, when invoked, has the effect of establishing a connection from a Student object to a specific Course object. Note that, in addition to having the target Student object as an implicit argument, the operation has an explicit argument called "course," which specifies the course for which the student wants to register. Explicit arguments are shown within parentheses.

Update operation: An operation that alters the state of an object.

A **scope operation** is an operation that applies to a class rather than an object instance. For example, avg_gpa for the Student class (not shown with the other operations for this class in Figure 14-2) calculates the average gpa across all students. (The operation name is underlined to indicate it is a scope operation.)

Scope operation: An operation that applies to a class rather than an object instance.

Representing Associations

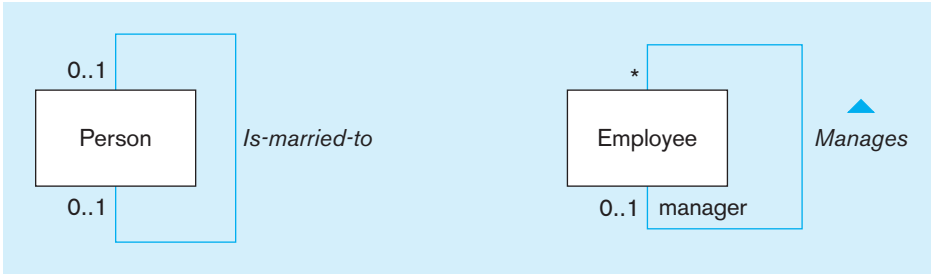
Association: A named relationship between or among object classes.

Parallel to the definition of a relationship for the E-R model, an **association** is a named relationship between or among instances of object classes. As in the E-R model, the degree of an association relationship may be one (unary), two (binary), three (ternary), or higher (*n*-ary). In Figure 14-3, we use examples from Figure 3-12 to illustrate how the object-oriented model can be used to represent association relationships of different degrees. An association is shown as a solid line between the participating classes. The end of an association where it connects to a class is called an **association role** (*UML Notation Guide*, 1997). Each association has two or more roles. A role may be explicitly named with a label near the end of an association (see the "manager" role in Figure 14-3a). The role name indicates the role played by the class attached to the end near which the name appears. Use of role names is optional. You can specify role names in place of, or in addition to, an association name.

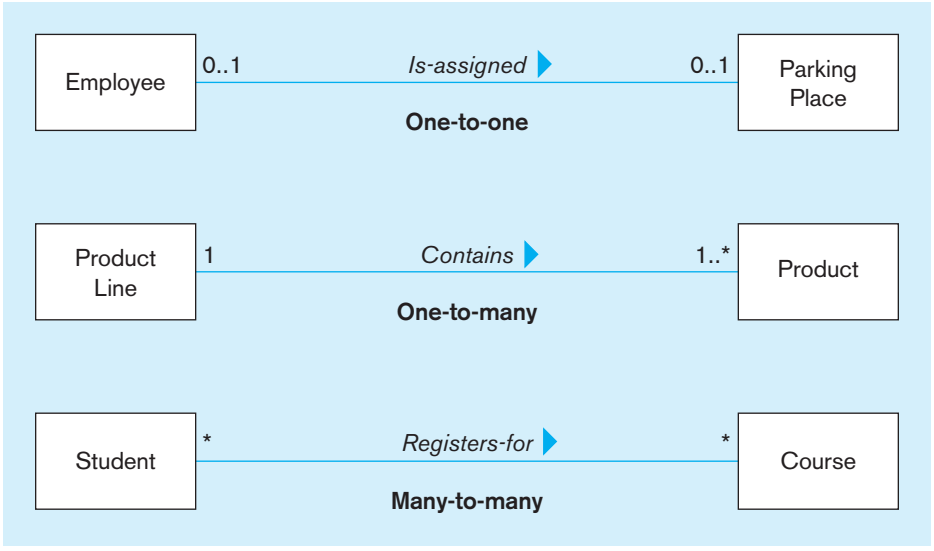
Association role: The end of an association where it connects to a class.

Figure 14-3a shows two unary relationships, *Is-married-to* and *Manages*. At one end of the *Manages* relationship, we have named the role as "manager," implying that an

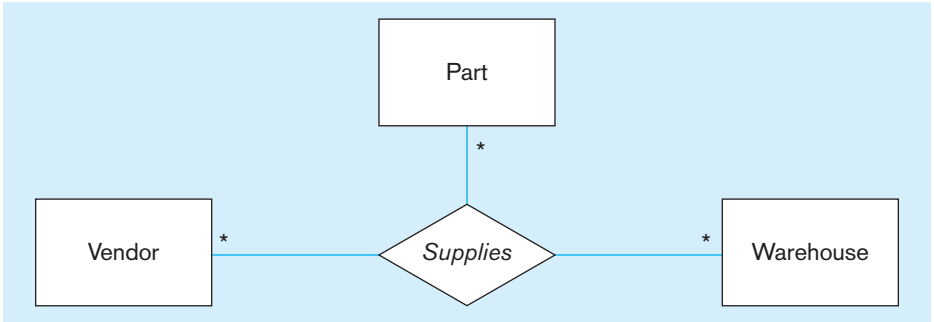
Figure 14-3
Examples of association relationships of different degrees
(a) Unary relationships



(b) Binary relationships



(c) Ternary relationship



employee can play the role of a manager. We have not named the other roles, but we have named the associations. When the role name does not appear, you may think of the role name as being that of the class attached to that end (Fowler, 2000). For example, the role for the right end of the *Is-assigned* relationship in Figure 14-3b could be called Parking Place.

Each role has a **multiplicity**, which indicates the number of objects that participate in a given relationship. In a class diagram, a multiplicity specification is shown as a text string representing an interval (or intervals) of integers in the following format:

Multiplicity: A specification that indicates how many objects participate in a given relationship.

lower-bound..upper-bound

The interval is considered to be closed, which means that the range includes both the lower and upper bounds. For example, a multiplicity of 2..5 denotes that a minimum of two and a maximum of five objects can participate in a given relationship. Multiplicities, therefore, are nothing but cardinality constraints that were discussed in Chapter 3. In addition to integer values, the upper bound of a multiplicity can be a star character (*), which denotes an infinite upper bound. If a single integer value is specified, it means that the range includes only that value.

The most common multiplicities in practice are 0..1, *, and 1. The 0..1 multiplicity indicates a minimum of zero and a maximum of one (optional one), whereas * (or equivalently, 0..*) represents the range from zero to infinity (optional many). A single 1 stands for 1..1, implying that exactly one object participates in the relationship (mandatory one).

The multiplicities for both roles in the *Is-married-to* relationship are 0..1, indicating that a person may be single or married to one person. The multiplicity for the manager role in the *Manages* relationship is 0..1 and that for the other role is *, implying that an employee may be managed by only one manager, but a manager may manage many employees.

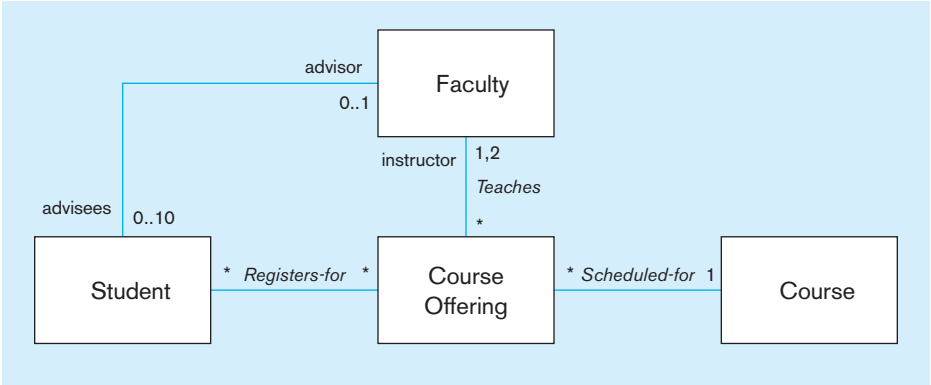
Figure 14-3b shows three binary relationships: *Is-assigned* (one-to-one), *Contains* (one-to-many), and *Registers-for* (many-to-many). A binary association is inherently bidirectional, though in a class diagram, the association name can be read in only one direction. For example, the *Contains* association is read from Product Line to Product. (Note: As in this example, you may show the direction explicitly by using a solid triangle next to the association name.) Implicit, however, is an inverse traversal of *Contains,* say, *Belongs-to,* which denotes that a product belongs to a particular product line. Both directions of traversal refer to the same underlying association; the name simply establishes a direction.

The diagram for the *Is-assigned* relationship shows that an employee is assigned a parking place or not assigned one at all (optional one). Reading in the other direction, we say that a parking place has either been allocated for a single employee or not allocated at all (optional one again). Similarly, we say that a product line contains many products, but at least one, whereas a given product belongs to exactly one product line (mandatory one). The diagram for the third binary association states that a student registers for multiple courses, but it is possible that he or she does not register at all, and a course, in general, has multiple students enrolled in it (optional many in both directions).

In Figure 14-3c, we show a ternary relationship called *Supplies* among Vendor, Part, and Warehouse. As in the E-R diagram, we represent a ternary relationship using a diamond symbol and place the name of the relationship there. The relationship is many-to-many-to-many, and, as discussed in Chapter 3, it cannot be replaced by three binary relationships without loss of information.

The class diagram in Figure 14-4a shows binary associations between Student and Faculty, between Course and Course Offering, between Student and Course Offering, and between Faculty and Course Offering. The diagram shows that a student may have an advisor, whereas a faculty member may advise up to a maximum of ten students. Also, although a course may have multiple offerings, a given course offering is scheduled for exactly one course.

Figure 14-4
Examples of binary association relationships
(a) University example



(b) Customer order example

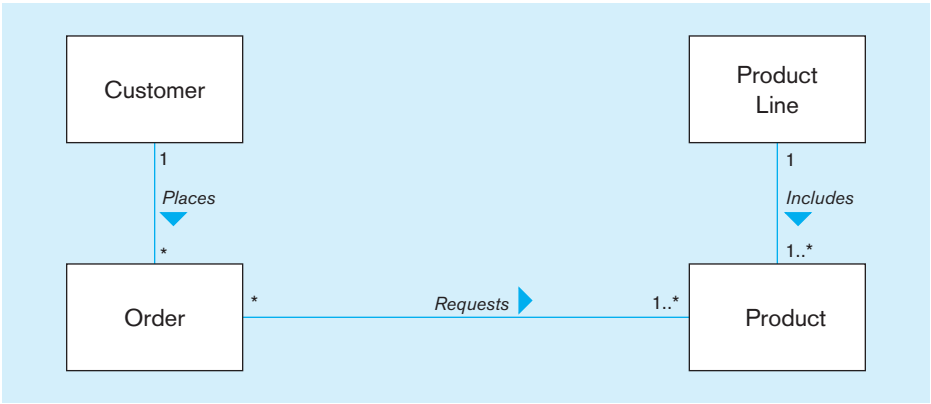


Figure 14-4a also shows that a faculty member plays the role of an instructor, as well as that of an advisor. Whereas the advisor role identifies the Faculty object associated with a Student object, the advisees role identifies the set of Student objects associated with a Faculty object. We could have named the association, say, Advises, but, in this case, the role names are sufficiently meaningful to convey the semantics of the relationship.

Figure 14-4b shows another class diagram for a customer order. The corresponding object diagram is presented in Figure 14-5; it shows some of the instances of the classes and the links among them. (Note: Just as an instance corresponds to a class, a link corresponds to a relationship.) In this example, we see the orders placed by two customers, Joe and Jane. Joe has placed two orders, Ord 20 and Ord 56. In Ord 20, Joe has ordered product P93 from the sports product line. In Ord 56, he has ordered the same sports product again, as well as product P50 from the hardware product line. Notice that Jane has ordered the same hardware product as Joe has, in addition to two other products (P9 and P10) from the cosmetics product line.

Representing Association Classes

Association class: An association that has attributes or operations of its own or that participates in relationships with other classes.

When an association itself has attributes or operations of its own, or when it participates in relationships with other classes, it is useful to model the association as an **association class** (just as we used an "associative entity" in Chapter 3). For example, in Figure 14-6a, the attributes term and grade really belong to the many-to-many association between Student and Course. The grade of a student for a course cannot be determined unless both the student and the course are known. Similarly, to find the term(s) in which the student took the course, both student and course must be known. The checkEligibility operation, which determines whether a student is eligible to register for a given course, also belongs to the association, rather than to any of

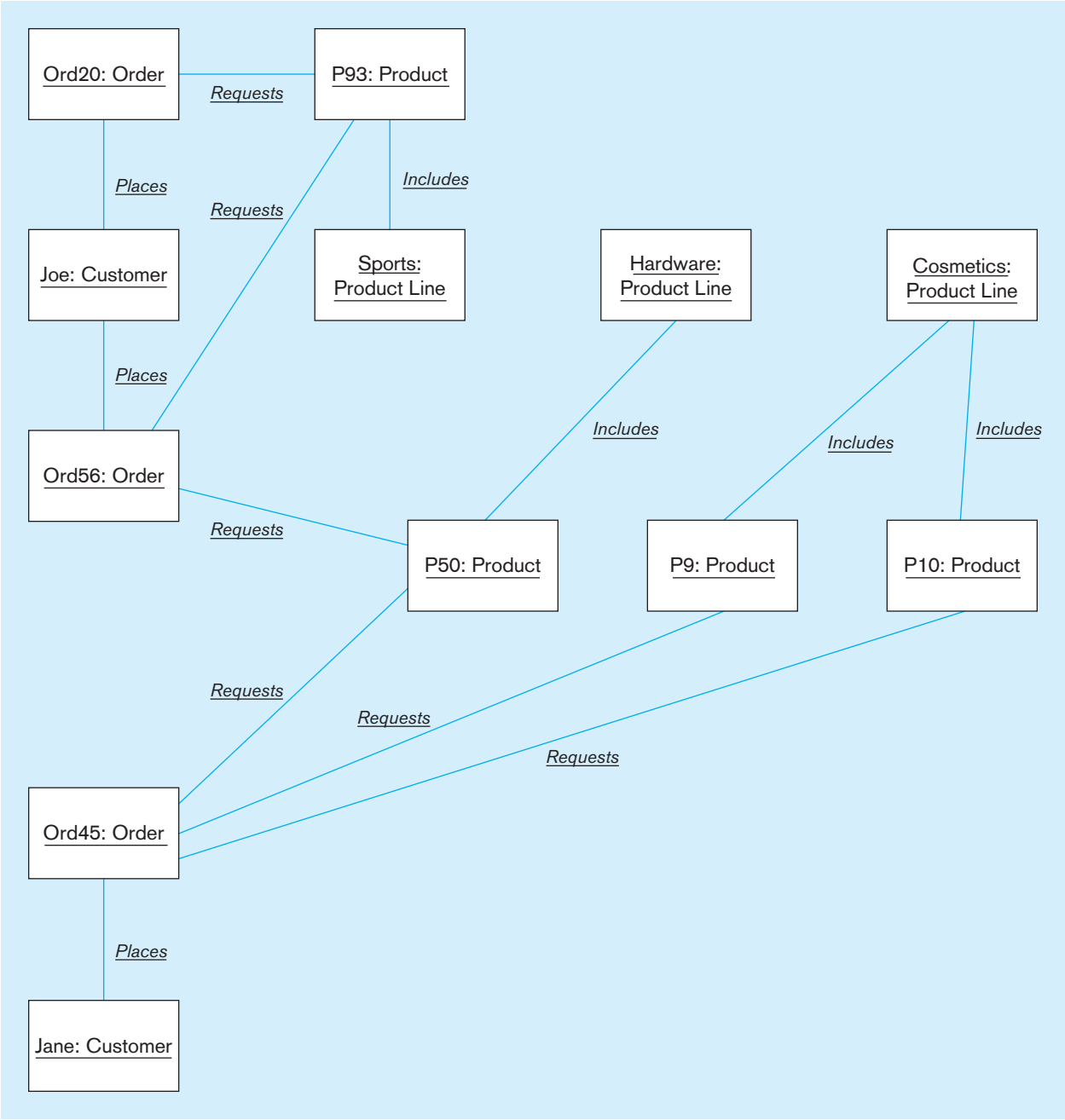
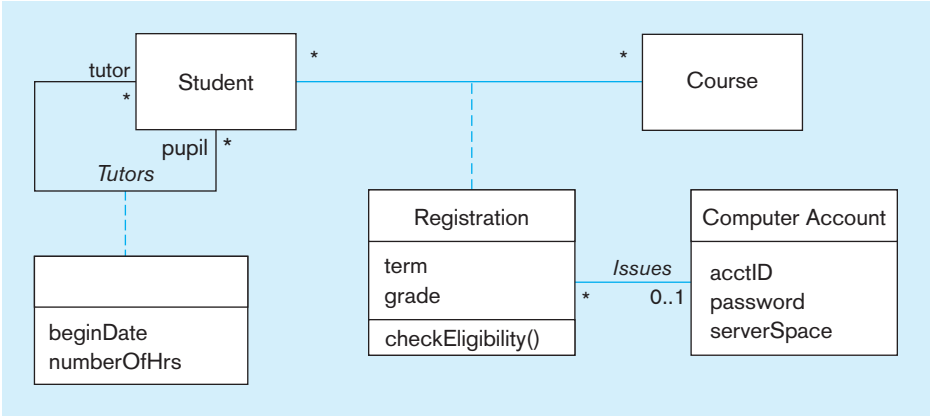


Figure 14-5
Object diagram for customer order example

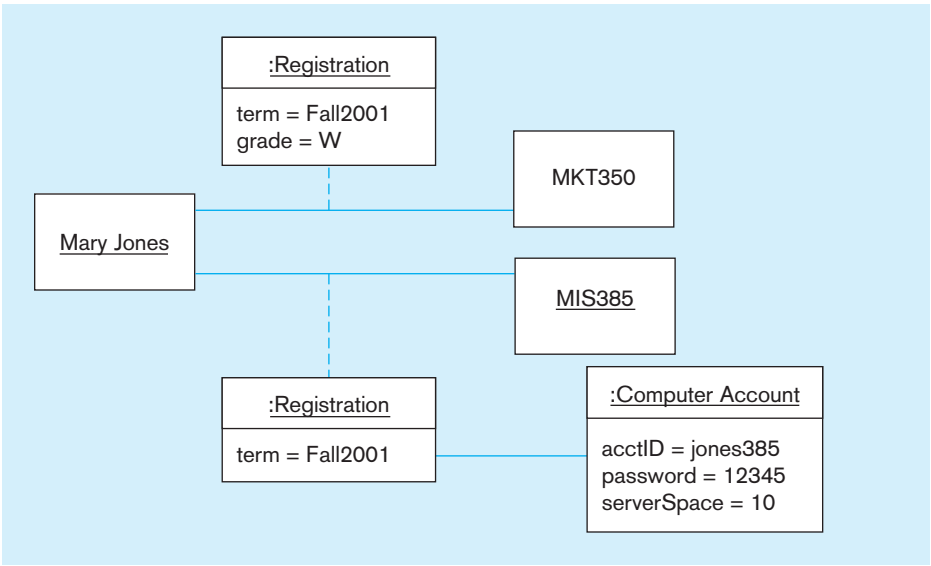
the two participating classes. We have also captured the fact that, for some course registrations, a computer account is issued to a student. For these reasons, we model Registration as an association class, having its own set of features and an association with another class (Computer Account). Similarly, for the unary *Tutors* association, beginDate and numberOfHrs (number of hours tutored) really belong to the association, and, therefore, appear in a separate association class.

You have the option of showing the name of an association class on the association path, or the class symbol, or both. When an association has only attributes, but does not have any operations or does not participate in other associations, the recommended option is to show the name on the association path, but to omit it from the association class symbol, to emphasize its "association nature" (*UML Notation Guide*, 1997). That is how we have shown the *Tutors* association. On the other hand,

Figure 14-6
Association class and link object
(a) Class diagram showing association classes



(b) Object diagram showing link objects



we have displayed the name of the Registration association—which has two attributes and one operation of its own, as well as an association called *Issues* with Computer Account—within the class rectangle to emphasize its "class nature."

Figure 14-6b shows a part of the object diagram representing a student, Mary Jones, and the courses she has registered for in the fall 2001 term: MKT350 and MIS385. Corresponding to an association class in a class diagram, link objects are present in an object diagram. In this example, there are two link objects (shown as :Registration) for the Registration association class, capturing the two course registrations. The diagram also shows that for the MIS385 course, Mary Jones has been issued a computer account with an ID, a password, and a designated amount of space on the server. She still has not received a grade for this course, but, for the MKT350 course, she received a grade of W because she withdrew from the course.

Figure 14-7 shows a ternary relationship among the Student, Software, and Course classes. It captures the fact that students use various software tools for different courses. For example, we could store the information that Mary Jones used Microsoft Access and Oracle for the Database Management course, Rational Rose and Visual C++ for the Object-Oriented Modeling course, and Level5 Object for the Expert Systems Course. Now suppose we want to estimate the number of hours per week Mary will spend using Oracle for the Database Management course. This process really belongs to the ternary association, and not to any of the individual classes. Hence, we have created an association class called Log, within which we have

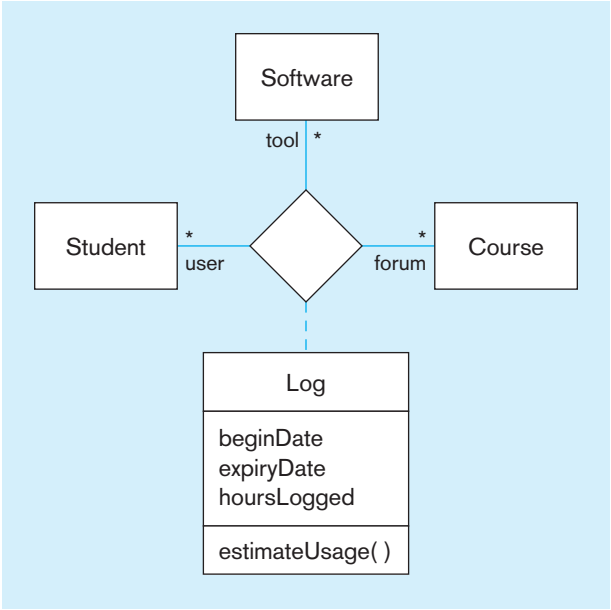


Figure 14-7
Ternary relationship with association class

declared an operation called estimateUsage. In addition to this operation, we have specified three attributes that belong to the association: beginDate, expiryDate, and hoursLogged.

REPRESENTING DERIVED ATTRIBUTES, DERIVED ASSOCIATIONS, AND DERIVED ROLES

A derived attribute, association, or role is one that can be computed or derived from other attributes, associations, and roles, respectively. (The concept of a derived attribute was introduced in Chapter 3.) A derived element (attribute, association, or role) is typically shown by placing either a slash (/) or a stereotype of <<Derived>> before the name of the element. For instance, in Figure 14-8, age is a derived attribute of Student, because it can be calculated from the date of birth and the current date. Because the calculation is a constraint on the object class, the calculation is shown on this diagram within {} above the Student object class. Also, the *Takes* relationship between Student and Course is derived, because it can be inferred from the *Registers-for* and *Scheduled-for* relationships. By the same token, participants is a derived role because it can be derived from other roles.

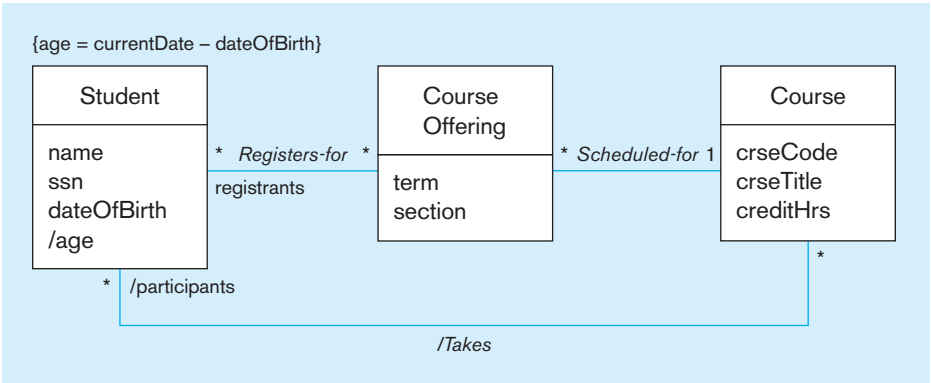
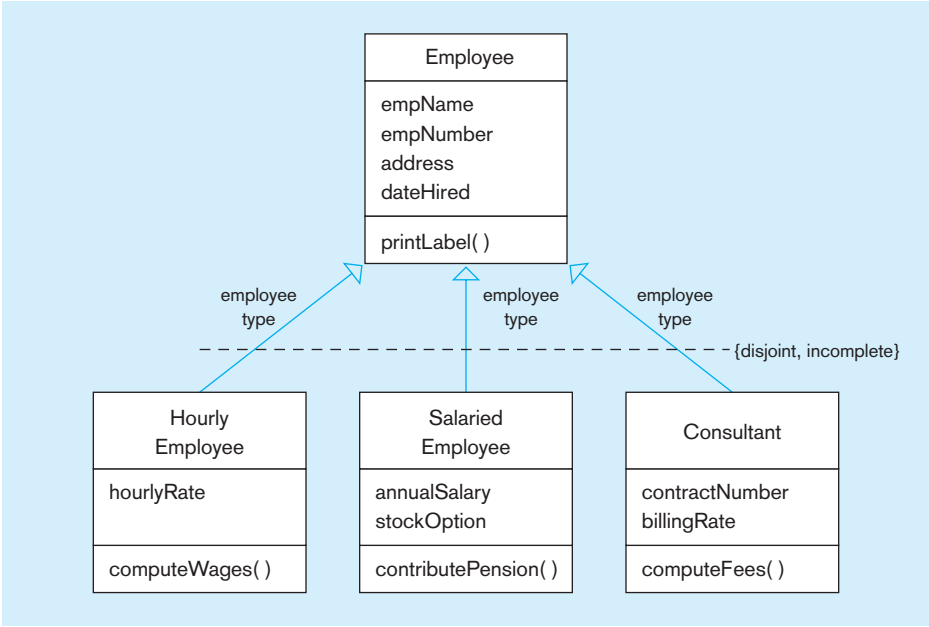


Figure 14-8
Derived attribute, association, and role

Representing Generalization

You were introduced to *generalization* and *specialization* in Chapter 4. Using the enhanced E-R model, you learned how to abstract the common attributes of two or more entities, as well as the common relationships in which they participate, into a more general entity supertype, while keeping the attributes and relationships that are not common in the entities (subtypes) themselves. In the object-oriented model, we apply the same notion, but with one difference. In generalizing a set of object classes into a more general class, we abstract not only the common attributes and relationships, but the common operations as well. The attributes and operations of a class are collectively known as the features of the class. The classes that are generalized are called subclasses, and the class they are generalized into is called a superclass, in perfect correspondence to subtypes and supertypes for EER diagramming.

Consider the example shown in Figure 14-9a. (See Figure 4-8 for the corresponding EER diagram.) There are three types of employees: hourly employees, salaried employees, and consultants. The features that are shared by all employees—empName, empNumber, address, dateHired, and printLabel—are stored in the



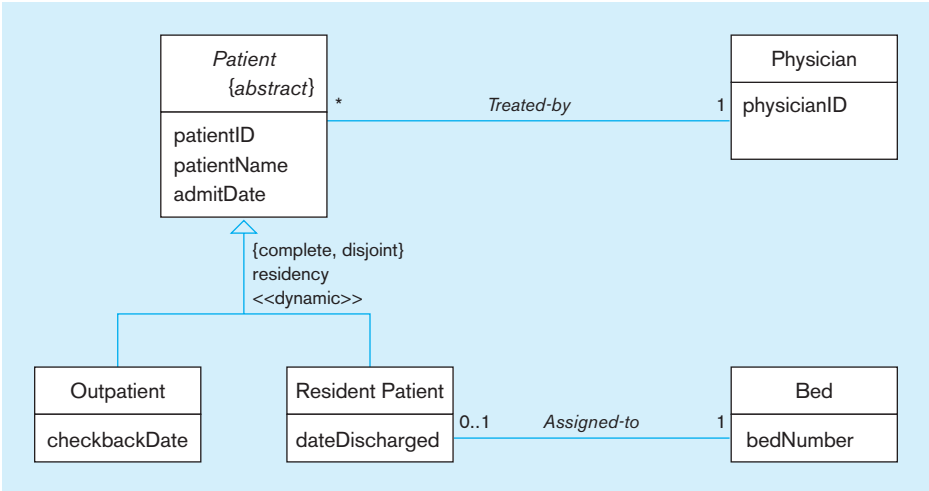


Figure 14-9
Examples of generalization, inheritance, and constraints
(a) Employee superclass with three subclasses

(b) Abstract Patient class with two concrete subclasses

Employee superclass, whereas the features that are peculiar to a particular employee type are stored in the corresponding subclass (e.g., hourlyRate and computeWages of Hourly Employee). A generalization path is shown as a solid line from the subclass to the superclass, with a hollow triangle at the end of, and pointing toward, the superclass. You can show a group of generalization paths for a given superclass as a tree with multiple branches connecting the individual subclasses, and a shared segment with a hollow triangle pointing toward the superclass. In Figure 14-9b (corresponding to Figure 4-3), for instance, we have combined the generalization paths from Outpatient to Patient, and from Resident Patient to Patient, into a shared segment with a triangle pointing toward Patient. We also specify that this generalization is dynamic, meaning that an object may change subtypes.

You can indicate the basis of a generalization by specifying a discriminator next to the path. A discriminator (corresponding to the subtype discriminator defined in Chapter 4) shows which property of an object class is being abstracted by a particular generalization relationship. You can discriminate on only one property at a time. For example, in Figure 14-9a, we discriminate the Employee class on the basis of employment type (hourly, salaried, consultant). To disseminate a group of generalization relationships as in Figure 14-9b, we need to specify the discriminator only once. Although we discriminate the Patient class into two subclasses, Outpatient and Resident Patient, based on residency, we show the discriminator label only once next to the shared line.

An instance of a subclass is also an instance of its superclass. For example in Figure 14-9b, an Outpatient instance is also a Patient instance. For that reason, a generalization is also referred to as an *Is-a* relationship. Also, a subclass inherits all the features from its superclass. For example, in Figure 14-9a, in addition to its own special features—hourlyRate and computeWages—the Hourly Employee subclass inherits empName, empNumber, address, dateHired, and printLabel from Employee. An instance of Hourly Employee will store values for the attributes of Employee and Hourly Employee, and, when requested, will apply the printLabel and computeWages operations.

Generalization and inheritance are transitive across any number of levels of a superclass/subclass hierarchy. For instance, we could have a subclass of Consultant called Computer Consultant that would inherit the features of Employee and Consultant. An instance of Computer Consultant would be an instance of Consultant and, therefore, an instance of Employee, too. Employee is an ancestor of Computer Consultant, while Computer Consultant is a descendant of Employee; these terms are used to refer to generalization of classes across multiple levels.

Inheritance is one of the major advantages of using the object-oriented model. It allows code reuse: There is no need for a programmer to write code that has already been written for a superclass. The programmer only writes code that is unique to the new, refined subclass of an existing class. In actual practice, object-oriented programmers typically have access to large collections of class libraries in their respective domains. They identify those classes that may be reused and refined to meet the demands of new applications. Proponents of the object-oriented model claim that code reuse results in productivity gains of several orders of magnitude.

Notice that in Figure 14-9b the *Patient* class is in italics, implying that it is an abstract class. An **abstract class** is a class that has no direct instances, but whose descendants may have direct instances (Booch, 1994; Rumbaugh et al., 1991). (Note: You can additionally write the word *abstract* within braces just below the class name. This is especially useful when you generate a class diagram by hand.) A class that can have direct instances (e.g., Outpatient or Resident Patient) is called a **concrete class**. In this example, therefore, Outpatient and Resident Patient can have direct instances, but *Patient* cannot have any direct instances of its own.

Abstract class: A class that has no direct instances, but whose descendants may have direct instances.

Concrete class: A class that can have direct instances.

The *Patient* abstract class participates in a relationship called Treated-by with Physician, implying that all patients—outpatients and resident patients alike—are

treated by physicians. In addition to this inherited relationship, the Resident Patient class has its own special relationship called *Assigned-to* with Bed, implying that only resident patients may be assigned to beds. So, in addition to refining the attributes and operations of a class, a subclass can also specialize the relationships in which it participates.

In Figures 14-9a and 14-9b, the words "complete," "incomplete," and "disjoint" have been placed within braces, next to the generalization. They indicate semantic constraints among the subclasses (complete corresponds to total specialization in the EER notation, whereas incomplete corresponds to partial specialization). In UML, a comma-separated list of keywords is placed either near the shared triangle, as in Figure 14-9b, or near a dashed line that crosses all of the generalization lines involved, as in Figure 14-9a (*UML Notation Guide,* 1997). Any of the following UML keywords may be used: overlapping, disjoint, complete, and incomplete. According to the *UML Notation Guide* (1997), these terms have the following meanings:

- *Overlapping* A descendant may be descended from more than one of the subclasses (same as the overlapping rule in EER diagramming).
- *Disjoint* A descendant may not be descended from more than one of the subclasses (same as the disjoint rule in EER diagramming).
- *Complete* All subclasses have been specified (whether or not shown). No additional subclasses are expected (same as the total specialization rule in EER diagramming).
- *Incomplete* Some subclasses have been specified, but the list is known to be incomplete. There are additional subclasses that are not yet in the model (same as the partial specialization rule in EER diagramming).

Both the generalizations in Figures 14-9a and 14-9b are disjoint. An employee can be an hourly employee, a salaried employee, or a consultant, but cannot, say, be both a salaried employee and a consultant at the same time. Similarly, a patient can be an outpatient or a resident patient, but not both. The generalization in Figure 14-9a is incomplete (a departure from what was shown in Figure 4-8), specifying that an employee might not belong to any of the three types. In such a case, an employee will be stored as an instance of Employee, a concrete class. In contrast, the generalization in Figure 14-9b is complete, implying that a patient has to be either an outpatient or a resident patient, and nothing else. For that reason, *Patient* has been specified as an abstract class.

In Figure 14-10, we show an example of an overlapping constraint. The diagram shows that research assistants and teaching assistants are graduate students. The overlapping constraint indicates that it is possible for a graduate student to serve as both a research assistant and a teaching assistant. For example, Sean Bailey, a graduate student, has a research assistantship of 12 hours/week and a teaching assistantship of 8 hours/week. Also notice that Graduate Student has been specified as a concrete class so that graduate students without an assistantship can be represented. The ellipsis (. . .) under the generalization line based on the "level" discriminator does not represent an incomplete constraint. It simply indicates that there are other subclasses in the model that have not been shown in the diagram. For example, although Undergrad Student is in the model, we have opted not to show it in the diagram since the focus is on assistantships. You may also use an ellipsis when there are space limitations.

In Figure 14-11, we represent both graduate and undergraduate students in a model developed for student billing. The calc-tuition operation computes the tuition a student has to pay; this sum depends on the tuition per credit hour (tuitionPerCred), the courses taken, and the number of credit hours (creditHrs) for each of those courses. The tuition per credit hour, in turn, depends on whether the student is a graduate or an undergraduate student. In this example, that amount is $300 for all graduate students and $250 for all undergraduate students. To denote

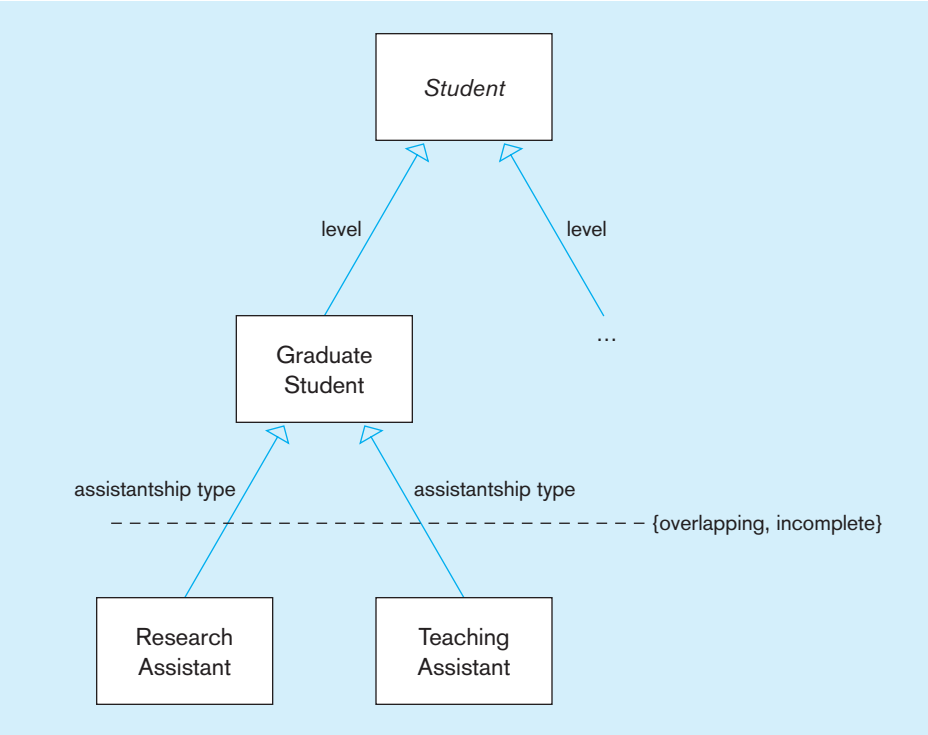


Figure 14-10
Example of overlapping constraint

that, we have underlined the tuitionPerCred attribute in each of the two subclasses, along with its value. Such an attribute is called a **class-scope attribute,** which specifies a value common to an entire class, rather than a specific value for an instance (Rumbaugh et al., 1991).

Class-scope attribute: An attribute of a class that specifies a value common to an entire class, rather than a specific value for an instance.

You can also specify an initial default value of an attribute using an "=" sign after the attribute name. This is the initial attribute value of a newly created object instance. For example, in Figure 14-11, the creditHrs attribute has an initial value of 3, implying that when a new instance of Course is created, the value of creditHrs is set to 3 by default. You can write an explicit constructor operation to modify the initial default value. The value may also be modified later through other operations. The difference between an initial value specification and a class-scope attribute is that while the former allows the possibility of different attribute values for the instances of a class, the latter forces all the instances to share a common value.

In addition to specifying the multiplicity of an association role, you can also specify other properties, for example, whether the objects playing the role are ordered. In the figure, we placed the keyword constraint "{ordered}" next to the Course Offering end of the *Scheduled-for* relationship to denote the fact that the offerings for a given course are ordered into a list—say, according to term and section. It is obvious that it makes sense to specify an ordering only when the multiplicity of the role is greater than one. The default constraint on a role is "{unordered}"; that is, if you do not specify the keyword "{ordered}" next to the role, it is assumed that the related elements form an unordered set. For example, the course offerings are not related to a student who registers for those offerings in any specific order.

The Graduate Student subclass specializes the abstract Student class by adding four attributes—undergradMajor, greScore, gmatScore, and tuitionPerCred—and by refining the inherited *calc-tuition* operation. Notice that the operation is shown in italics within the Student class, indicating that it is an abstract operation. An **abstract operation** defines the form or protocol of the operation, but not its implementation (Rumbaugh et al., 1991). In this example, the Student class defines the protocol of the *calc-tuition* operation, without providing the corresponding **method** (the actual

Abstract operation: Defines the form or protocol of the operation, but not its implementation.

Method: The implementation of an operation.

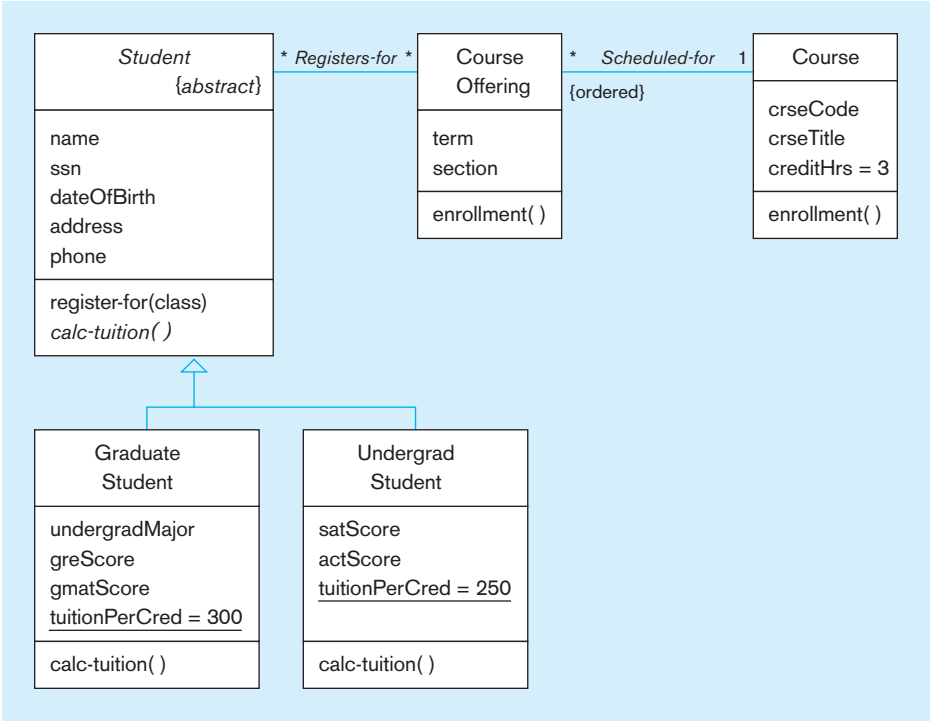


Figure 14-11
Polymorphism, abstract operation, class-scope attribute, and ordering

implementation of the operation). The protocol includes the number and types of the arguments, the result type, and the intended semantics of the operation. The two concrete subclasses, Graduate Student and Undergrad Student, supply their own implementations of the calc-tuition operation. Note that because these classes are concrete, they cannot store abstract operations.

It is important to note that although the Graduate Student and Undergraduate Student classes share the same calc-tuition operation, they might implement the operation in quite different ways. For example, the method that implements the operation for a graduate student might add a special graduate fee for each course the student takes. The fact that the same operation may apply to two or more classes in different ways is known as polymorphism, a key concept in object-oriented systems. The enrollment operation in Figure 14-11 illustrates another example of **polymorphism**. While the enrollment operation within Course Offering computes the enrollment for a particular course offering or section, an operation with the same name within Course computes the combined enrollment for all sections of a given course.

Polymorphism: The same operation may apply to two or more classes in different ways.

Interpreting Inheritance and Overriding

We have seen how a subclass can augment the features inherited from its ancestors. In such cases, the subclass is said to use *inheritance for extension*. On the other hand, if a subclass constrains some of the ancestor attributes or operations, it is said to use *inheritance for restriction* (Booch, 1994; Rumbaugh et al., 1991). For example, a subclass called Tax-Exempt Company may suppress or block the inheritance of an operation called compute-tax from its superclass, Company.

The implementation of an operation can also be overridden. **Overriding** is the process of replacing a method inherited from a superclass by a more specific implementation of that method in a subclass. The reasons for overriding include extension, restriction, and optimization (Rumbaugh et al., 1991). The name of the new operation remains the same as the inherited one, but it has to be explicitly shown within the subclass to indicate that the operation is overridden.

Overriding: The process of replacing a method inherited from a superclass by a more specific implementation of that method in a subclass.

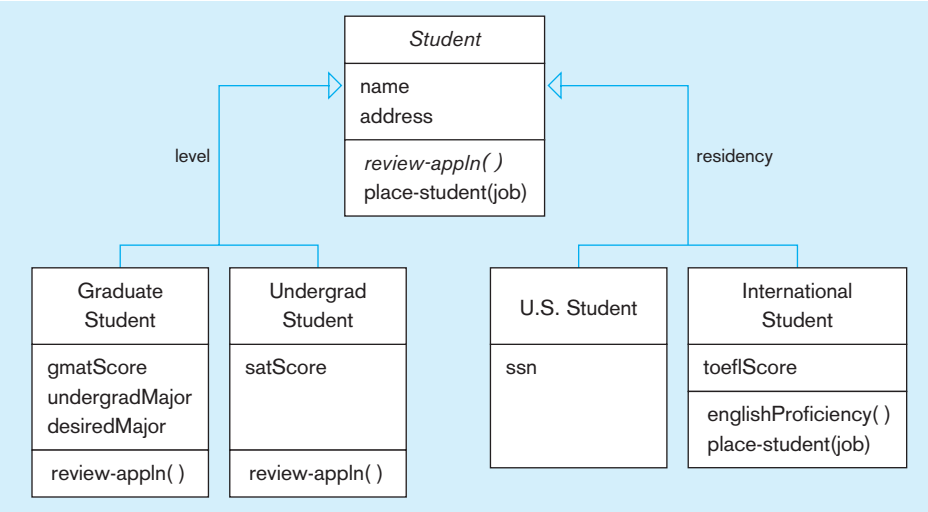


Figure 14-12
Overriding inheritance

In *overriding for extension,* an operation inherited by a subclass from its superclass is extended by adding some behavior (code). For example, a subclass of Company called Foreign Company inherits an operation called compute-tax but extends the inherited behavior by adding a foreign surcharge to compute the total tax amount.

In *overriding for restriction,* the protocol of the new operation in the subclass is restricted. For example, an operation called place-student(job) in Student may be restricted in the International Student subclass by tightening the argument job (see Figure 14-12). While students in general may be placed in all types of jobs during the summer, international students may be limited to only on-campus jobs because of visa restrictions. The new operation overrides the inherited operation by tightening the job argument, restricting its values to only a small subset of all possible jobs. This example also illustrates the use of multiple discriminators. While the basis for one set of generalizations is a student's "level" (graduate or undergraduate), that for the other set is his or her "residency" status (U.S. or international).

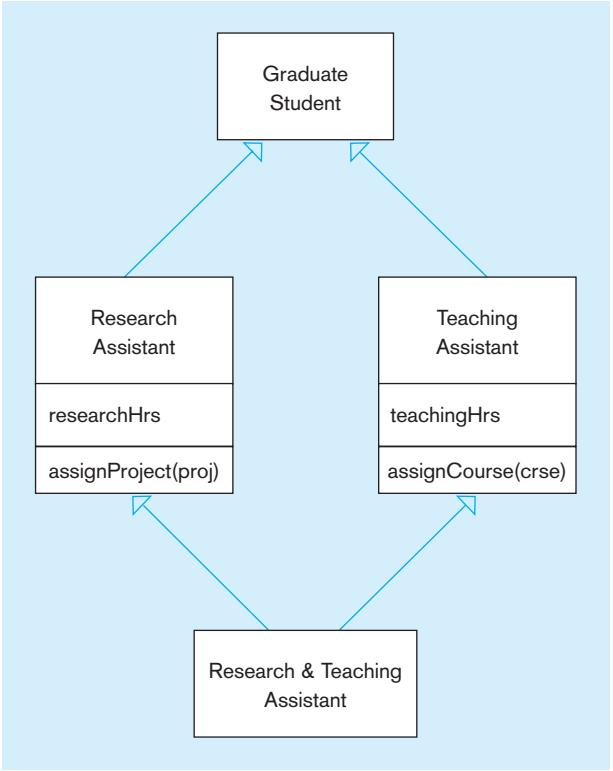
In *overriding for optimization,* the new operation is implemented with improved code by exploiting the restrictions imposed by a subclass. Consider, for example, a subclass of Student called Dean's List Student, which represents all those students who are on the Dean's List. To qualify for the Dean's List, a student must have a grade point average greater than or equal to 3.50. Suppose Student has an operation called mailScholApps, which mails applications for merit- and means-tested scholarships to students who have a gpa greater than or equal to 3.00, and whose family's total gross income is less than $20,000. The method for the operation in Student will have to check the conditions, whereas the method for the same operation in the Dean's List Student subclass can improve upon the speed of execution by removing the first condition from its code. Consider another operation called findMinGpa, which finds the minimum gpa among the students. Suppose the Dean's List Student class is sorted in ascending order of the gpa, but the Student class is not. The method for findMinGpa in Student must perform a sequential search through all the students. In contrast, the same operation in Dean's List Student can be implemented with a method that simply retrieves the gpa of the first student in the list, thereby obviating the need for a time-consuming search.

Representing Multiple Inheritance

So far you have been exposed to single inheritance, where a class inherits from only one superclass. But sometimes, as we saw in the example with research and teaching assistants, an object may be an instance of more than one class. This is known as **multiple classification** (Fowler, 2000; *UML Notation Guide,* 1997). For

Multiple classification: An object is an instance of more than one class.

Figure 14-13
Multiple inheritance



instance, Sean Bailey, who has both types of assistantships, has two classifications: one as an instance of Research Assistant, and the other as an instance of Teaching Assistant. Multiple classification, however, is discouraged by experts. It is not supported by the ordinary UML semantics and many object-oriented languages.

To get around the problem, we can use multiple inheritance, which allows a class to inherit features from more than one superclass. For example, in Figure 14-13, we have created Research & Teaching Assistant, which is a subclass of both Research Assistant and Teaching Assistant. All students who have both research and teaching assistantships may be stored under the new class. We may now represent Sean Bailey as an object belonging to only the Research & Teaching Assistant class, which inherits its features from both its parents, such as researchHrs and assignProject(proj) from Research Assistant and teachingHrs and assignCourse(crse) from Teaching Assistant (and provides no unique features of its own).

Representing Aggregation

Aggregation: A part-of relationship between a component object and an aggregate object.

An **aggregation** expresses a *Part-of* relationship between a component object and an aggregate object. It is a stronger form of association relationship (with the added "part-of" semantics) and is represented with a hollow diamond at the aggregate end. For example, Figure 14-14 shows a personal computer as an aggregate of CPU (up to four for multiprocessors), hard disks, monitor, keyboard, and other objects (a typical bill-of-materials structure). Note that aggregation involves a set of distinct object instances, one of which contains or is composed of the others. For example, a Personal Computer object is related to (consists of) one to four CPU objects, one of its parts. As shown in Figure 14-14, it is also possible for component objects to exist without being part of a whole (e.g., there can be a Monitor that is not part of any PC). Further, it is possible that the Personal Computer class has operations that apply to its parts; for example, calculating the extended warranty cost for the PC involved an analysis of its component parts. In contrast, generalization relates object classes:

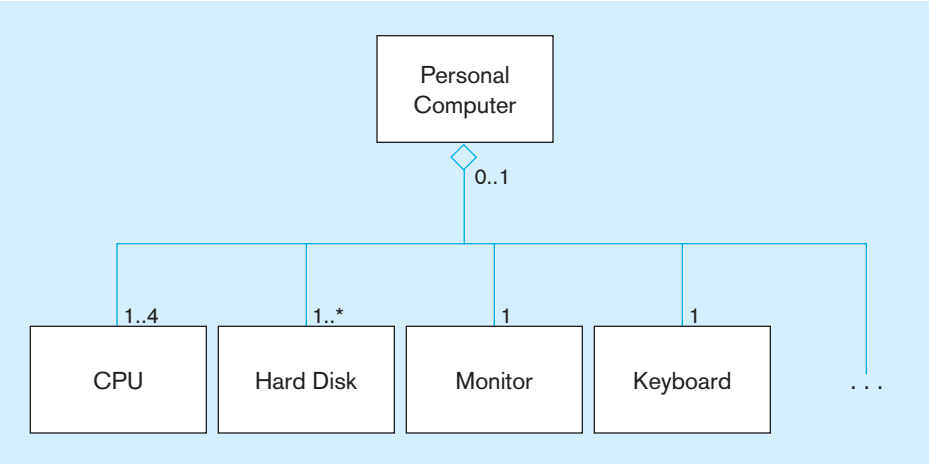


Figure 14-14
Example of aggregation

an object (e.g., Mary Jones) is simultaneously an instance of its class (e.g., Undergrad Student) and its superclass (e.g., Student). Only one object (e.g., Mary Jones) is involved in a generalization relationship. This is why multiplicities are indicated at the ends of aggregation lines, whereas there are no multiplicities for generalization relationships.

Figure 14-15a shows an aggregation structure of a university. The object diagram in Figure 14-15b shows how Riverside University, a University object instance, is related to its component objects, which represent administrative units (e.g., Admissions, Human Resources, etc.) and schools (e.g., Arts and Science, Business, etc.). A school object (e.g., Business), in turn, comprises several department objects (e.g., Accounting, Finance, etc.).

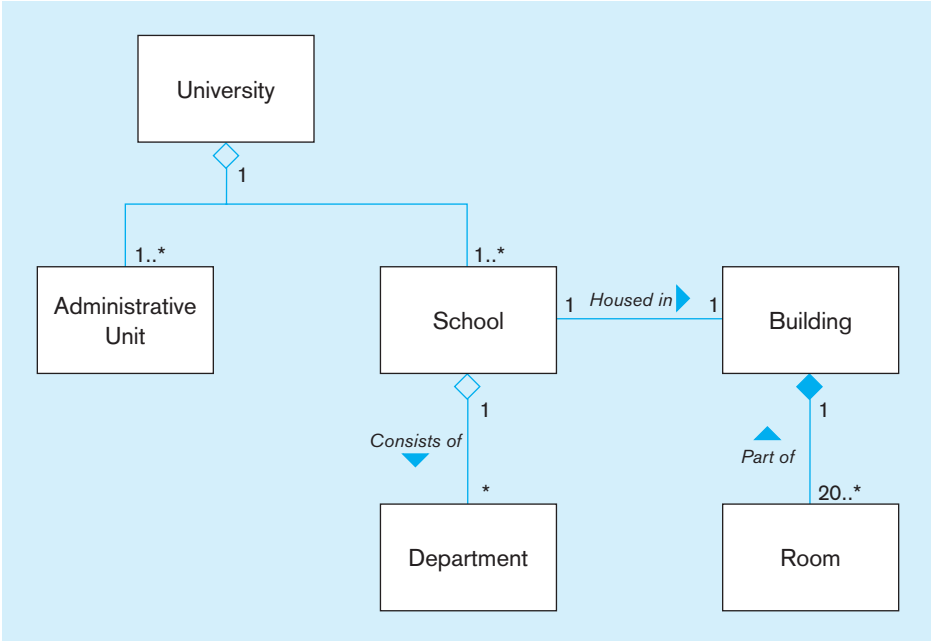
Notice that the diamond at one end of the relationship between Building and Room is not hollow, but solid. A solid diamond represents a stronger form of aggregation, known as composition (Fowler, 2000). In **composition,** a part object belongs to one and only one whole object; for example, a room is part of only one building and cannot exist by itself. Therefore, the multiplicity on the aggregate end is exactly one. Parts may be created after the creation of the whole object; for example, rooms may be added to an existing building. However, once a part of a composition is created, it lives and dies with the whole; deletion of the aggregate object cascades to its components. If a building is demolished, for example, so are all its rooms. However, it is possible to delete a part before its aggregate dies, just as it is possible to demolish a room without bringing down a building.

Composition: A part object that belongs to only one whole object and that lives and dies with the whole object.

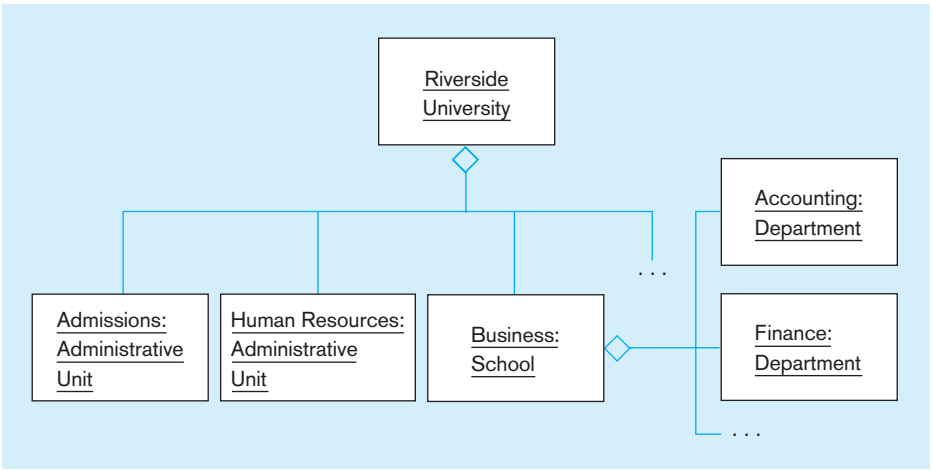
Consider another example of aggregation, the bill-of-materials structure presented earlier in Chapter 3. Many manufactured products are made up of assemblies, which in turn are composed of subassemblies and parts, and so on. We saw how we could represent this type of structure as a many-to-many unary relationship (called Has_components) in an E-R diagram (see Figure 3-13). When the relationship has an attribute of its own, such as Quantity, the relationship can be converted to an associative entity. Note that although the bill-of-materials structure is essentially an aggregation, we had to represent it as an association because the E-R model does not support the semantically stronger concept of aggregation. In the object-oriented model, we can explicitly show the aggregation.

In Figure 14-16, we have represented the bill-of-materials structure. To distinguish between an assembly and a primitive part (one without components), we have created two classes, Assembly and Simple Part, both of which are subclasses of a class called Part. The diagram captures the fact that a product consists of many parts, which themselves can be assemblies of other parts, and so on; this is an example of recursive aggregation. Because Part is represented as an abstract class, a part is either

Figure 14-15
Aggregation and composition
(a) Class diagram



(b) Object diagram



an assembly or a primitive part. An Assembly object is an aggregate of instances of the Part superclass, implying that it is composed of other assemblies (optional) and primitive parts. Note that we can easily capture an attribute, such as the quantity of parts in an assembly, inside an association class attached to the aggregation relationship.

When you are unsure whether a relationship between two objects is an association or an aggregation, try to figure out if one object is really part of the other object. That is, is there a whole/part relationship? Note that an aggregation does not necessarily have to imply physical containment, such as that between Personal Computer and CPU. The whole/part relationship may be conceptual, for example, the one between a mutual fund and a certain stock that is part of the fund. In an aggregation, an object may or may not exist independently of an aggregate object. For example, a stock exists irrespective of whether it is part of a mutual fund or not, while a department does not exist independently of an organization. Also, an object may be part of several aggregate objects (e.g., many mutual funds may contain IBM stocks in their portfolios). Remember, however, that while this is possible in aggregation, composition does not allow an object to be part of more than one aggregate object.

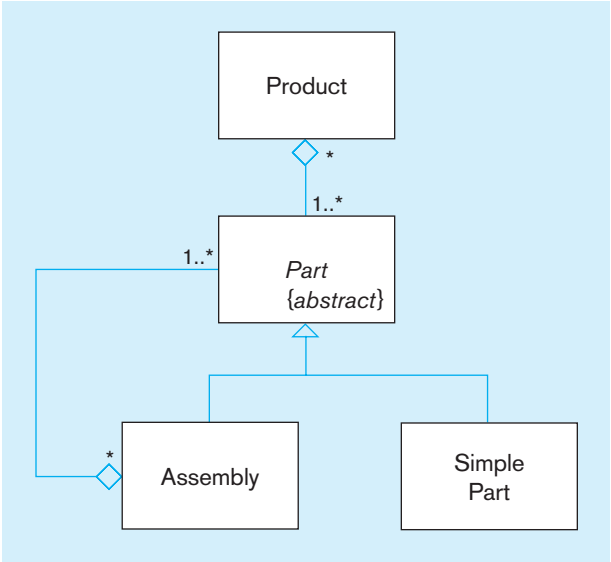


Figure 14-16
Recursive aggregation

Another characteristic of aggregation is that some of the operations on the whole automatically apply to its parts. For example, an operation called ship in the Personal Computer object class applies to CPU, Hard Disk, Monitor, and so on because whenever a computer is shipped, so are its parts. The ship operation on Personal Computer is said to *propagate* to its parts (Rumbaugh et al., 1991).

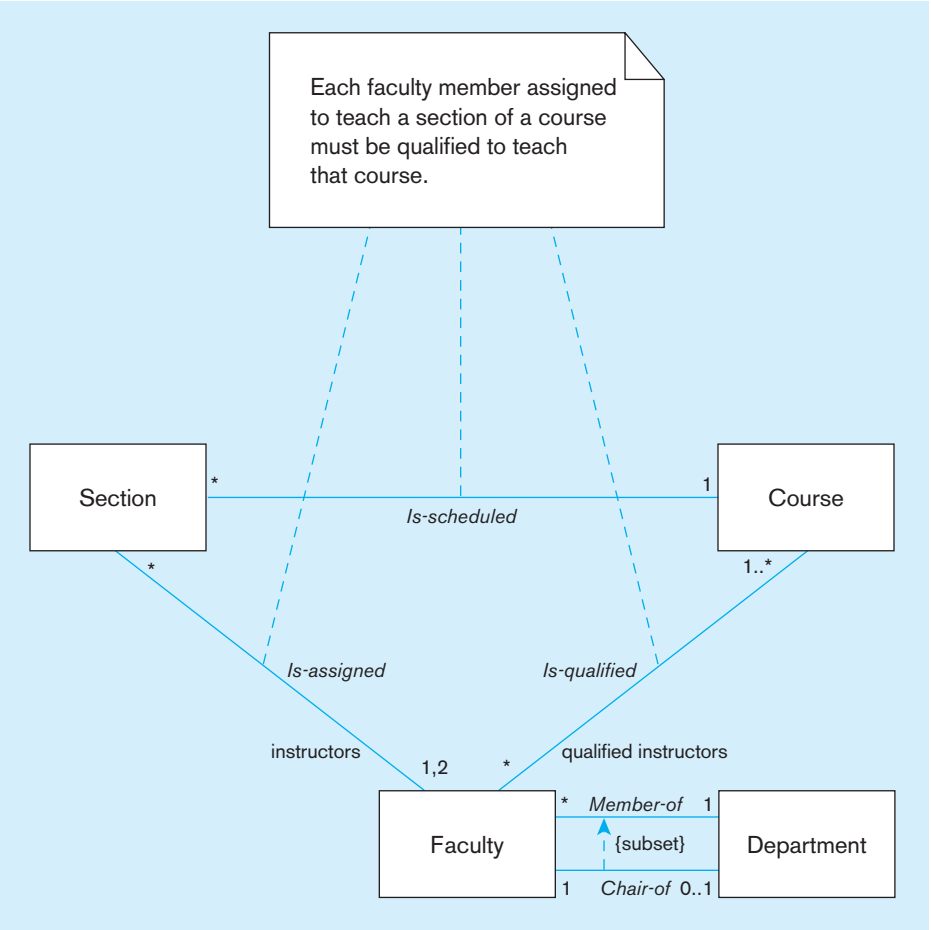
BUSINESS RULES

Business rules were discussed in detail earlier in Chapters 3 and 4. You saw how to express different types of rules in an E-R diagram. In the examples provided in this chapter, we have captured many business rules as constraints—implicitly, as well as explicitly—on classes, instances, attributes, operations, relationships, and so on. For example, you saw how to specify cardinality constraints and ordering constraints on association roles. You also saw how to represent semantic constraints (e.g., overlapping, disjoint, etc.) among subclasses. Many of the constraints that have been discussed so far in this chapter were imposed by including a set of UML keywords within braces, for example, {disjoint, complete} and {ordered}, and placing them close to the elements to which the constraints apply. For example, in Figure 14-11, we expressed a business rule that offerings for a given course are ordered. But if you cannot represent a business rule using such a predefined UML constraint, you can define the rule in plain English or in some other language such as formal logic.

When you have to specify a business rule involving two graphical symbols (such as those representing two classes or two associations), you can show the constraint as a dashed arrow from one element to the other, labeled by the constraint name in braces (*UML Notation Guide,* 1997). In Figure 14-17, for example, we have stated the business rule that the chair of a department must be a member of the department by specifying the *Chair-of* association as a subset of the *Member-of* association.

When a business rule involves three or more graphical symbols, you can show the constraint as a note and attach the note to each of the symbols by a dashed line (*UML Notation Guide,* 1997). In Figure 14-17, we have captured the business rule that "each faculty member assigned to teach a section of a course must be qualified to teach that course" within a note symbol. Because this constraint involves all

Figure 14-17
Representing business rules



three association relationships, we have attached the note to each of the three association paths.

OBJECT MODELING EXAMPLE: PINE VALLEY FURNITURE COMPANY

In Chapters 3 and 4, you saw how to develop a high-level E-R diagram for the Pine Valley Furniture Company (see Figures 3-22 and 4-12). We identified the entity types, as well as their keys and other important attributes, based on a study of the business processes at the company. We will now show you how to develop a class diagram for the same application using the object-oriented approach. The class diagram is shown in Figure 14-18. We discuss the commonalities, as well as the differences, between this diagram and the E-R diagrams in the prior figures. Figure 14-18 is based primarily on Figure 4-12, but the attributes from Figure 3-22 are now also included. Figure 14-18 is developed using the UML drawing tool in Microsoft Visio 2002. Another very popular UML tool is Rational Rose (see Quatrani, 2000).

As you would expect, the entity types are represented as object classes and all the attributes are shown within the classes. Note, however, that you do not need to show explicit identifiers in the form of primary keys, because, by definition, each object has its own identity. The E-R model, as well as the relational data model (see Chapter 5), requires you to specify explicit identifiers because there is no other way of supporting the notion of identity. In the object-oriented model, the only identifiers you should

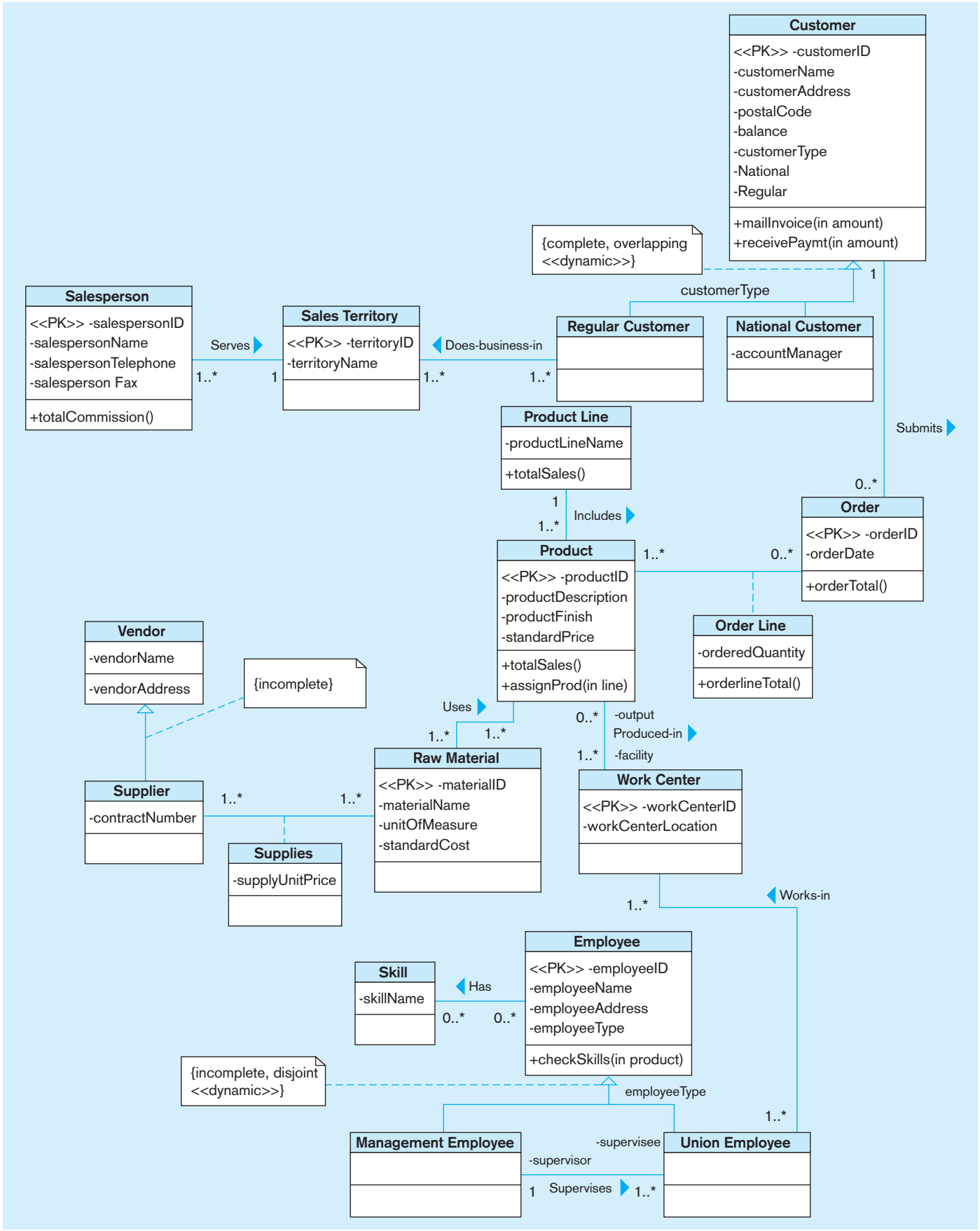


Figure 14-18
Class diagram for
Pine Valley Furniture Company

represent are attributes that make sense in the real world, such as salespersonID, customerID, orderID, and productID, which are represented as primary keys (the <<PK>> stereotype) in the class diagram. Notice that we have not shown an identifier for Product Line, based on the assumption that Product_Line_ID was merely included in the E-R diagram as an internal identifier, not as a real-world attribute, such as orderID or productID. If Pine Valley Furniture Company does not actually use vendorID or, for that matter, any other attribute, to support its business processes, you should not include that attribute in the class diagram. For that reason, we have not shown identifiers for classes such as Vendor, Order Line, and Skill.

Role names are applied to some relationships. For example, Product plays the role of output and Work Center plays the role of facility in the Produced-in relationship.

The class diagram of Figure 14-18 shows several processes, represented as operations, that could not be captured in an E-R diagram. For example, Customer has an operation called mailInvoice, which, when executed, mails an invoice to a customer who has placed an order, specifying the total order amount in dollars, and increases the customer's outstanding balance by that amount. On receipt of payment from the customer, the receivePaymt operation adjusts the balance by the amount received. The orderlineTotal operation of Order Line computes the total dollar amount for a given order line of an order, whereas the orderTotal operation of Order computes the total amount for an entire order (i.e., the sum total of the amounts on all the order lines).

Figure 14-18 also illustrates polymorphism. The totalSales operation appears within both the Product and Product Line classes, but is implemented as two different methods. While the method in Product computes the total sales for a given product, the one in Product Line computes the total sales of all products belonging to a given product line.

The following operations—totalSales, totalCommission, orderTotal, orderlineTotal, and checkSkills—are all query operations, because they do not have any side effects. In contrast, mailInvoice, receivePaymt, and assignProd are all update operations because they alter the state of some object(s). For example, the assignProd operation assigns a new product to the product line specified in the "line" argument, thereby changing the state of both the product, which becomes assigned, and the product line, which includes one more product.

Specifications for the generalizations are shown in constraint boxes. So, for example, there are no other Customer types than Regular Customer and National Customer (complete constraint), a customer can be simultaneously of both types (overlapping constraint), and a customer can switch between subtypes (<<dynamic>> stereotype). Customers are distinguished by the value of customerType. Customer is an abstract class because of the complete constraint.

Summary

In this chapter, we introduced the object-oriented modeling approach, which is becoming increasingly popular because it supports effective representation of a real-world application—both in terms of its data and processes—using a common underlying representation. We described the activities involved in the different phases of the object-oriented development life cycle and emphasized the seamless nature of the transitions that an object-oriented model undergoes as it evolves through the different phases, from analysis to design to implementation. This is in sharp contrast to other modeling approaches, such as structured analysis and design, which lack a common underlying representation and, therefore, suffer from abrupt and disjoint model transitions.

We presented object-oriented modeling as a high-level conceptual activity, especially as it pertains to data analysis. We introduced the concept of objects and classes and discussed object identity and encapsulation. Throughout the chapter, we developed several class diagrams, using the UML notation, to show you how to model various types of situations. You also learned how to draw an object diagram that corresponds to a given class

diagram. The object diagram is an instance of the class diagram, providing a snapshot of the actual objects and links present in a system at some point in time.

We showed how to model the processes underlying an application using operations. We discussed three types of operations: constructor, query, and update. The E-R model (as well as the enhanced E-R model) does not allow you to capture processes; it allows you only to model the data needs of an organization. In this chapter, we emphasized several similarities between the E-R model and the object-oriented model, but, at the same time, highlighted those features that make the latter more powerful than the former.

We showed how to represent association relationships of different degrees—unary, binary, and ternary—in a class diagram. An association has two or more roles; each role has a multiplicity, which indicates the number of objects that participate in the relationship. Other types of constraints can be specified on association roles, such as forming an ordered set of objects. When an association itself has attributes or operations of its own, or when it participates in other associations, the association is modeled as a class; such a class is called an association class. Links and link objects in an object diagram correspond to associations and association classes, respectively, in a class diagram. Derived attributes, derived relationships, and derived roles can also be represented in a class diagram.

The object-oriented model expresses generalization relationships using superclasses and subclasses, similar to supertypes and subtypes in the enhanced E-R model. The basis of a generalization path can be denoted using a discriminator label next to the generalization path. Semantic constraints among subclasses can be specified using UML keywords such as overlapping, disjoint, complete, and incomplete. When a class does not have any direct instances, it is modeled as an abstract class. An abstract class may have an abstract operation, whose form, but not method, is provided.

In a generalization relationship, a subclass inherits features from its superclass, and by transitivity, from all its ancestors. Inheritance is a very powerful mechanism because it supports code reuse in object-oriented systems. We discussed ways of applying inheritance of features, as well as reasons for overriding inheritance of operations in subclasses. We also introduced another key concept in object-oriented modeling, that of polymorphism, which means that an operation can apply in different ways across different classes. The concepts of encapsulation, inheritance, and polymorphism in object-oriented modeling provide system developers with powerful mechanisms for developing complex, robust, flexible, and maintainable business systems.

The object-oriented model supports aggregation, whereas the E-R or the enhanced E-R model does not. Aggregation is a semantically stronger form of association, expressing the Part-of relationship between a component object and an aggregate object. We distinguished between aggregation and generalization and provided you with tips for choosing between association and aggregation in representing a relationship. We discussed a stronger form of aggregation, known as composition, in which a part object belongs to only one whole object, living and dying together with it.

In this chapter, you also learned how to state business rules implicitly, as well as explicitly, in a class diagram. UML provides several keywords that can be used as constraints on classes, attributes, relationships, and so on. In addition, user-defined constraints may be used to express business rules. When a business rule involves two or more elements, you saw how to express the rule in a class diagram, such as by using a note symbol. We concluded the chapter by developing a class diagram for Pine Valley Furniture Company, illustrating how to apply the object-oriented approach to model both the data and the processes underlying real-world business problems.

CHAPTER REVIEW

Key Terms

Abstract class
Abstract operation
Aggregation
Association
Association class
Association role
Behavior
Class
Class diagram
Class-scope attribute
Composition
Concrete class
Constructor operation
Encapsulation
Method
Multiple classification
Multiplicity
Object
Object diagram
Operation
Overriding
Polymorphism
Query operation
Scope operation
State
Update operation

Review Questions

1. Define each of the following terms:
   a. class
   b. state
   c. behavior
   d. encapsulation
   e. operation
   f. method
   g. constructor operation
   h. query operation
   i. update operation
   j. abstract class
   k. concrete class
   l. abstract operation
   m. multiplicity
   n. class-scope attribute
   o. association class
   p. polymorphism
   q. overriding
   r. multiple classification
   s. composition
   t. recursive aggregation

2. Match the following terms to the appropriate definitions:

____ concrete class	a. operation applied in different ways
____ abstract operation	b. form, not implementation
____ aggregation	c. direct instances
____ overriding	d. belongs to only one whole object
____ polymorphism	e. method replacement
____ association class	f. part-of relationship
____ composition	g. a set of objects
____ class	h. equivalent to associative entity

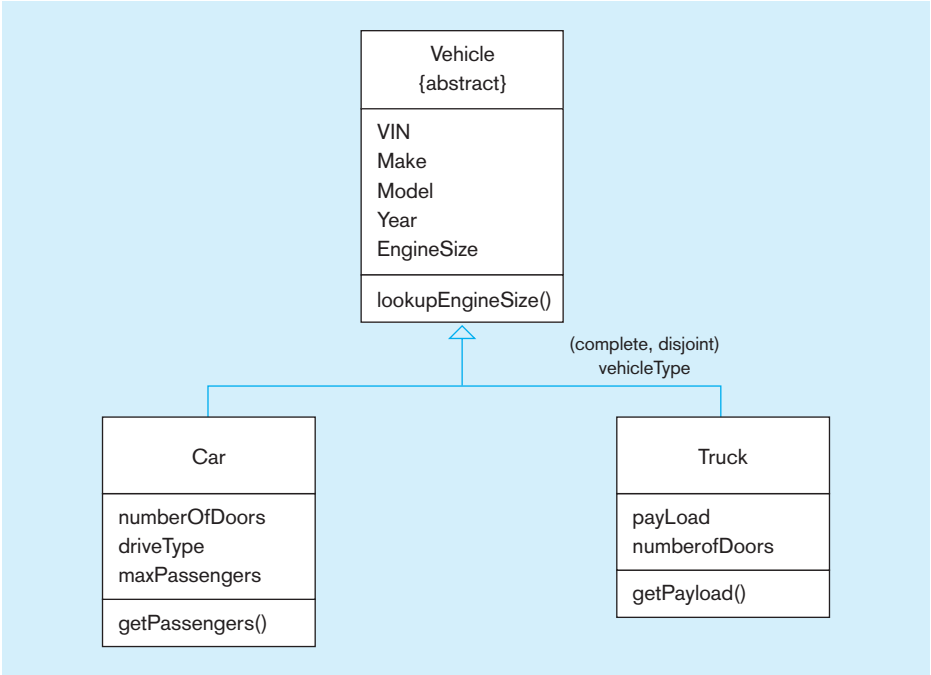
3. Contrast the following terms:
   a. class; object
   b. attribute; operation
   c. state; behavior
   d. operation; method
   e. query operation; update operation
   f. abstract class; concrete class
   g. class diagram; object diagram
   h. association; aggregation
   i. generalization; aggregation
   j. aggregation; composition
   k. overriding for extension; overriding for restriction

4. State the activities involved in each of the following phases of the object-oriented development life cycle: object-oriented analysis, object-oriented design, and object-oriented implementation.

5. Compare and contrast the object-oriented model with the enhanced E-R model.

6. State the conditions under which a designer should model an association relationship as an association class.

Figure 14-19



7. Using a class diagram, give an example for each of the following types of relationships: unary, binary, and ternary. Specify the multiplicities for all the relationships.
8. Add role names to the association relationships you identified in Review Question 7.
9. Add operations to some of the classes you identified in Review Question 7.
10. Give an example of generalization. Your example should include at least one superclass and three subclasses and a minimum of one attribute and one operation for each of the classes. Indicate the discriminator and specify the semantic constraints among the subclasses.
11. If the diagram you developed for Review Question 10 does not contain an abstract class, extend the diagram by adding an abstract class that contains at least one abstract operation. Also, indicate which features of a class are inherited by other classes.
12. Using (and, if necessary, extending) the diagram from your solution to Review Question 11, give an example of polymorphism.
13. Give an example of aggregation. Your example should include at least one aggregate object and three component objects. Specify the multiplicities at each end of all of the aggregation relationships.
14. What makes the object-oriented modeling approach a powerful tool for developing complex systems?
15. Given the class diagram shown in Figure 14-19, can we have an instance of Vehicle? Why or why not?
16. Why does the UML provide several different types of diagrams?
17. In the diagram shown in Figure 14-20, what do we call the Assignment class?
18. When would a unary relationship need to be represented as an association class?
19. In the class diagram shown in Figure 14-21, what do we call /AvailBalance? What do we call /purchases? Why are these used in this diagram?

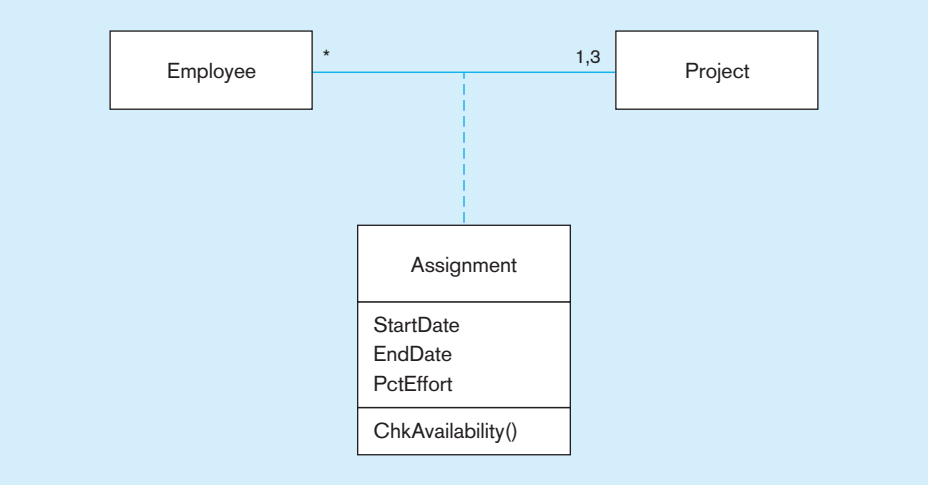


Figure 14-20

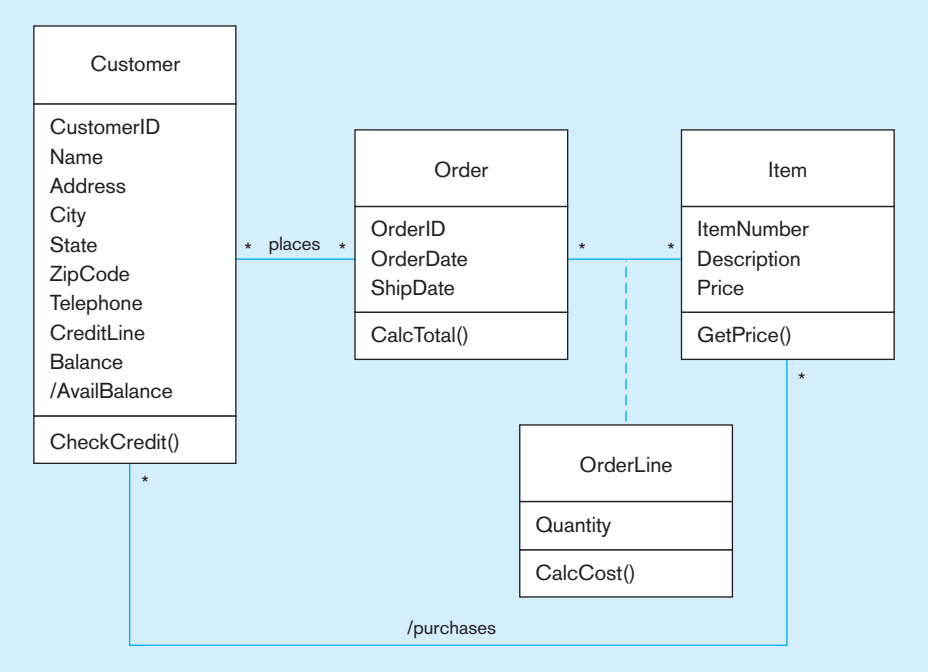
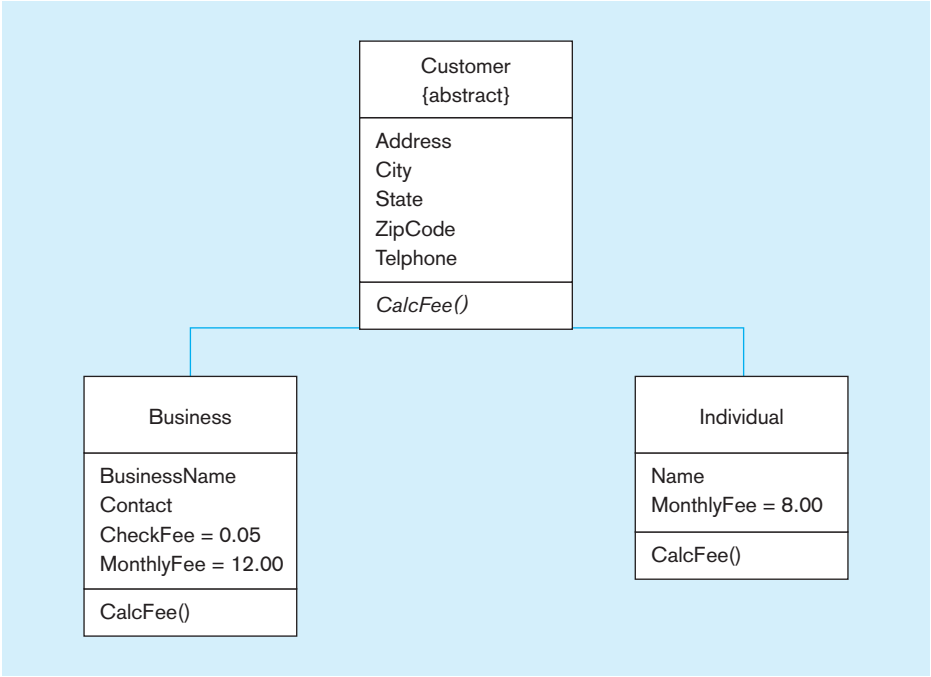


Figure 14-21

Figure 14-22



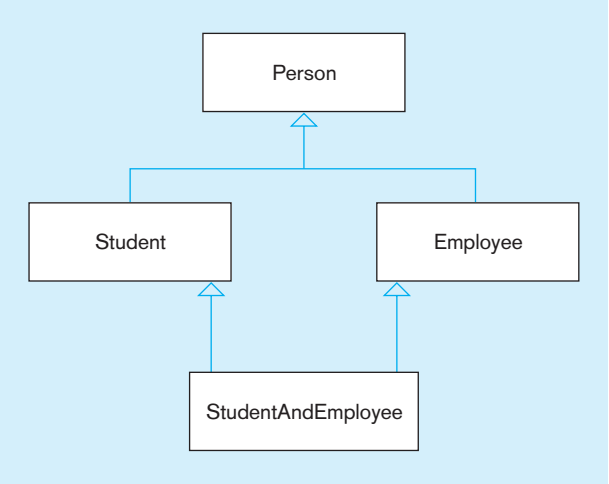


Figure 14-23

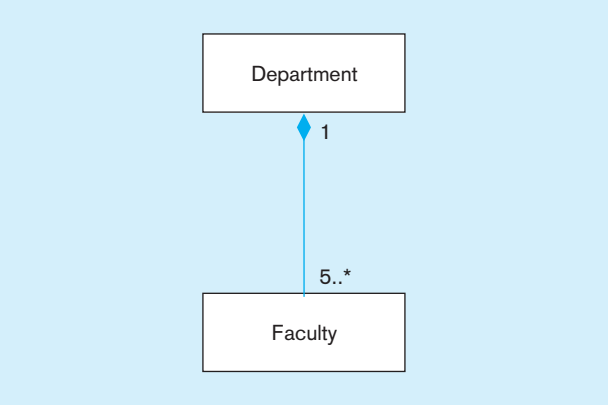


Figure 14-24

20. In the class diagram shown in Figure 14-22, CheckFee and MonthlyFee are examples of _____ _____ attributes. What type of operation is CalcFee?
21. The class diagram shown in Figure 14-23 is an example of _____.
22. The class diagram shown in Figure 14-24 is an example of _____. Is the relationship between faculty and their department represented properly in this diagram? Why or why not?

Problems and Exercises

1. Draw a class diagram, showing the relevant classes, attributes, operations, and relationships, for each of the following situations (if you believe that you need to make additional assumptions, clearly state them for each situation):

   a. A company has a number of employees. The attributes of Employee include employeeID (primary key), name, address, and birthdate. The company also has several projects. Attributes of Project include projectName and

startDate. Each employee may be assigned to one or more projects or may not be assigned to a project. A project must have at least one employee assigned and may have any number of employees assigned. An employee's billing rate may vary by project, and the company wishes to record the applicable billing rate for each employee when assigned to a particular project. At the end of each month, the company mails a check to each employee who has worked on a project during that month. The amount of the check is based on the billing rate and the hours logged for each project assigned to the employee.

b. A university has a large number of courses in its catalog. Attributes of Course include courseNumber (primary key), courseName, and units. Each course may have one or more different courses as prerequisites or may have no prerequisites. Similarly, a particular course may be a prerequisite for any number of courses or may not be prerequisite for any other course. The university adds or drops a prerequisite for a course only when the director for the course makes a formal request to that effect.

c. A laboratory has several chemists who work on one or more projects. Chemists also may use certain kinds of equipment on each project. Attributes of Chemist include name and phoneNo. Attributes of Project include projectName and startDate. Attributes of Equipment include serialNo and cost. The organization wishes to record assignDate—that is, the date when a given equipment item was assigned to a particular chemist working on a specified project—as well as totalHours—that is, the total number of hours the chemist has used the equipment for the project. The organization also wants to track the usage of each type of equipment by a chemist. It does so by computing the average number of hours the chemist has used that equipment on all assigned projects. A chemist must be assigned to at least one project and one equipment item. A given equipment item need not be assigned, and a given project need not be assigned either a chemist or an equipment item.

d. A college course may have one or more scheduled sections, or may not have a scheduled section. Attributes of Course include courseID, courseName, and units. Attributes of Section include sectionNumber and semester. The value of sectionNumber is an integer (such as "1" or "2") that distinguishes one section from another for the same course, but does not uniquely identify a section. There is an operation called findNumSections that finds the number of sections offered for a given course in a given semester.

e. A hospital has a large number of registered physicians. Attributes of Physician include physicianID (primary key) and specialty. Patients are admitted to the hospital by physicians. Attributes of Patient include patientID (primary key) and patientName. Any patient who is admitted must have exactly one admitting physician. A physician may optionally admit any number of patients. Once admitted, a given patient must be treated by at least one physician. A particular physician may treat any number of patients, or may not treat any patients. Whenever a patient is treated by a physician, the hospital wishes to record the details of the treatment, by including the date, time, and results of the treatment.

2. A student, whose attributes include studentName, address, phone, and age, may engage in multiple campus-based activities. The university keeps track of the number of years a given student has participated in a specific activity and, at the end of each academic year, mails an activity report to the student showing his participation in various activities. Draw a class diagram for this situation.

3. Prepare a class diagram for a real estate firm that lists property for sale. The following describes this organization:

- The firm has a number of sales offices in several states; location is an attribute of sales office.
- Each sales office is assigned one or more employees. Attributes of employee include employeeID and employee Name. An employee must be assigned to only one sales office.
- For each sales office, there is always one employee assigned to manage that office. An employee may manage only the sales office to which he or she is assigned.
- The firm lists property for sale. Attributes of property include propertyName and location.
- Each unit of property must be listed with one (and only one) of the sales offices. A sales office may have any number of properties listed or may have no properties listed.
- Each unit of property has one or more owners. Attributes of owner are ownerName and address. An owner may own one or more units of property. For each property that an owner owns, an attribute called percentOwned indicates what percentage of the property is owned by the owner.

Add a subset constraint between two of the associations you identified in your class diagram.

4. Draw a class diagram for some organization that you are familiar with—Boy Scouts/Girl Scouts, sports team, and so on. In your diagram, indicate names for at least four association roles.

5. Draw a class diagram for the following situation (state any assumptions you believe you have to make in order to develop a complete diagram): Stillwater Antiques buys and sells one-of-a-kind antiques of all kinds (e.g., furniture, jewelry , china, and clothing). Each item is uniquely identified by an item number and is also characterized by a description, asking price, condition, and open-ended comments. Stillwater works with many different individuals, called clients, who sell items to and buy items from the store. Some clients only sell items to Stillwater, some only buy items, and some others both sell and buy. A client is identified by a client number and is also described by a client name and client address. When Stillwater sells an item in stock to a client, the owners want to record the commission paid, the actual selling price, sales tax (tax of zero indicates a tax exempt sale), and date sold. When Stillwater buys an item from a client, the owners want to record the purchase cost, date purchased, and condition at time of purchase.

6. Draw a class diagram for the following situation (state any assumptions you believe you have to make in order to develop a complete diagram): The H. I. Topi School of Business operates international business programs in ten locations throughout Europe. The school had its first class of 9,000 graduates in 1965. The School keeps track of each graduate's student number, name, country of birth, current country of citizenship, current name, current address, and the name of each major the student completed. (Each student has one or two majors.) To maintain strong ties to its alumni, the school holds various events around the world. Events have a title, date, location, and type (e.g., reception, dinner, or seminar). The school needs to keep track of which graduates have attended which events. For an attendance by a graduate at an event, a comment is recorded about information school officials learned from that graduate at that event. The school also keeps in contact with graduates by mail, e-mail, telephone, and fax interactions. As with events, the school records information learned from the graduate from each of these contacts. When a school official knows that he or she will be meeting or talking to a graduate, a report is produced showing the latest information about that graduate and the information learned during the past two years from that graduate from all contacts and events the graduate attended.

7. Draw a class diagram for the following problem. A nonprofit organization depends on a number of different types of persons for its successful operation. The organization is interested in the following attributes for all of these persons: Social Security number, name, address, and phone. There are three types of persons who are of greatest interest: employees, volunteers, and donors. In addition to the attributes for a person, an employee has an attribute called dateHired, and a volunteer has an attribute called skill. A donor is a person who has donated one or more items to the organization. An item, specified by a name, may have no donors, or one or more donors. When an item is donated, the organization records its price, so that at the end of the year, it can identify the top ten donors.

   There are persons other than employees, volunteers, and donors who are of interest to the organization, so a person need not belong to any of these three groups. On the other hand, at a given time a person may belong to two or more of these groups (e.g., employee and donor).

8. A bank has three types of accounts: checking, savings, and loan. Following are the attributes for each type of account:

   CHECKING: Acct_No, Date_Opened, Balance, Service_Charge

   SAVINGS: Acct_No, Date_Opened, Balance, Interest_Rate

   LOAN: Acct_No, Date_Opened, Balance, Interest_Rate, Payment

   Assume that each bank account must be a member of exactly one of these subtypes. At the end of each month, the bank computes the balance in each account and mails a statement to the customer holding that account. The balance computation depends on the type of the account. For example, a checking account balance may reflect a service charge, whereas a savings account balance may include an interest amount. Draw a class diagram to represent the situation. Your diagram should include an abstract class, as well as an abstract operation for computing the balance.

9. Refer to the class diagram for hospital relationships (Figure 14-9b). Add notation to express the following business rule: A resident patient can be assigned a bed only if that patient has been assigned a physician who will assume responsibility for the patient's care.

10. An organization has been entrusted with developing a Registration and Title system that maintains information about all vehicles registered in a particular state. For each vehicle that is registered with the office, the system has to store the name, address, and telephone number of the owner, the start date and end date of the registration, plate information (issuer, year, type, and number), sticker (year, type, and number), and registration fee. In addition, the following information is maintained about the vehicles themselves: the number, year, make, model, body style, gross weight, number of passengers, diesel-powered (yes/no), color, cost, and mileage. If the vehicle is a trailer, the parameters diesel-powered and number of passengers are not relevant. For travel trailers, the body number and length must be known. The system needs to maintain information on the luggage capacity for a car, maximum cargo capacity and maximum towing capacity for a truck, and horsepower for a motorcycle. The system issues registration notices to owners of vehicles whose registrations are due to expire after two months. When the owner renews the registration, the system updates the registration information on the vehicle.

    a. Develop an object-oriented model by drawing a class diagram that shows all the object classes, attributes, operations, relationships, and multiplicities. For each operation, show its argument list.

    b. Each vehicle consists of a drive train, which, in turn, consists of an engine and a transmission. (Ignore the fact that a trailer doesn't have an engine and a transmission.) Suppose that, for each vehicle, the system has to maintain the following information: the size and number of cylinders of its engine and the type and weight of its transmission. Add classes, attributes, and relationships to the class diagram to capture this new information.

    c. Give a realistic example (you may create one) of an operation that you override in a subclass or subclasses. Add the operation at appropriate places in the class diagram and discuss the reasons for overriding.

11. Each semester, each student must be assigned an adviser who counsels students about degree requirements and helps students register for classes. Each student must register for classes with the help of an adviser, but if the student's assigned adviser is not available, the student may register with any adviser. We must keep track of students, the assigned adviser for each, and the name of the adviser with whom the student registered for the current term. Represent this situation of students and advisers with an E-R diagram. Also draw a data model for this situation using the tool you have been told to use in your course.

12. Draw a class diagram, showing the relevant classes, attributes, operations, and relationships for the following situation: Emerging Electric wishes to create a database with the following classes and attributes:

- Customer, with attributes Customer_ID, Name, Address (Street, City, State, Zip code), Telephone
- Location, with attributes Location_ID, Address (Street, City, State, Zip code), Type (Business or Residential)
- Rate, with attributes RateClass, Rateperkwh

After interviews with the owners, you have come up with the following business rules:

- Customers can have one or more locations.
- Each location can have one or more rates, depending upon the time of day.

13. Draw a class diagram, showing the relevant classes, attributes, operations, and relationships for the following situation: Wally Los Gatos, owner of Wally's Wonderful World of Wallcoverings, has hired you as a consultant to design a database management system for his chain of three stores that sell wallpaper and accessories. He would like to track sales, customers, and employees. After an initial meeting with Wally, you have developed the following list of business rules and specifications:

- Customers place orders through a branch.
  - Wally would like to track the following about customers: Name, Address, City, State, Zip code, Telephone, Date of Birth, Primary Language
  - A customer may place many orders.
  - A customer does not always have to order through the same branch all the time.
  - Customers may have one or more accounts, although they may also have no accounts.
  - The following information needs to be recorded about accounts: Balance, Last payment date, Last payment amount, Type
- A branch may have many customers.
  - The following information about each branch needs to be recorded: Branch number, Location (Address, City, State, Zip code), Square footage
  - A branch may sell all items, or may only sell certain items.
- Orders are composed of one or more items.
  - The following information about each order needs to be recorded: Order date, Credit authorization status
- Items may be sold by one or more branches.
  - We wish to record the following about each item: Description, Color, Size, Pattern, Type
- An item can be composed of multiple items; for example, a dining room wallcovering set (item 20) may consist of wallpaper (item 22) and borders (item 23).
- Wally employs fifty-six employees. He would like to track the following information about employees: Name, Address (Street, City, State, Zip code), Telephone, Date of Hire, Title, Salary, Skill, Age
  - Each employee works in one and only one branch.
  - Each employee may have one or more dependents. We wish to record the name of the dependent as well as the age and relationship.
  - Employees can have one or more skills.

Please indicate any assumptions that you have made.

14. Our friend Wally Los Gatos (see Problem and Exercise 13), realizing that his wallcovering business had a few wrinkles in it, decided to pursue a law degree at night. After graduating, he has teamed up with Lyla El Pàjaro to form Peck and Paw, Attorneys at Law. Wally and Lyla have hired you to design a database system based on the set of business rules defined below. It is in your best interest to perform a thorough analysis, in order to avoid needless litigation. Please create a class diagram based upon the following set of rules.

- An ATTORNEY is retained by one or more CLIENTS for each CASE.
- Attributes of ATTORNEY are: Attorney_ID, Name, Address, City, State, Zip code, Specialty (may be more than one), Bar (may be more than one)
- A CLIENT may have more than one ATTORNEY for each CASE.
- Attributes of CLIENT are: Client_ID, Name, Address, City, State, Zip code, Telephone, Date of Birth
- A CLIENT may have more than one CASE.
- Attributes of CASE are: Case_ID, Case_Description, Case_Type
- An ATTORNEY may have more than one CASE.
- Each CASE is assigned to one and only one COURT.
- Attributes of COURT are: Court_ID, Court_Name, City, State, Zip code
- Each COURT has one or more JUDGES assigned to it.
- Attributes of JUDGE are: Judge_ID, Name, Years in Practice
- Each JUDGE is assigned to exactly one court.

15. Draw a class diagram, showing the relevant classes, attributes, operations, and relationships for the following situation: An international school of technology has hired you to create a database management system in order to assist in scheduling classes. After several interviews with the president, you have come up with the following list of classes, attributes, and initial business rules:

Room

Attributes: Building_ID, Room_No, Capacity
Room is identified by Building_ID and Room_NO.
A room can be either a lab or a classroom. If it is a classroom, it has an additional attribute called board type.

MediaType

Attributes: Mtypeid (identifying attribute), TypeDescription

Please note: We are tracking the type of media (such as a VCR, projector, etc.), not the individual piece of equipment. Tracking of equipment is outside of the scope of this project.

ComputerType

Attributes: Ctypeid (identifying attribute), TypeDescription, DiskCapacity, ProcessorSpeed

Please note: As with MediaType, we are tracking only the type of computer, not an individual computer. You can think of this as a class of computers (e.g., PIII 900MHZ).

Instructor

Attributes: Emp_ID (identifying attribute), Name, Rank, Office_Phone

Time slot

Attributes: TSID (identifying attribute), DayofWeek, StartTime, EndTime

Course

Attributes: CourseID (identifying attribute), CourseDescription, Credits

Courses can have one, none, or many prerequisites.
Courses also have one or more sections. Section has the following attributes: SectionID, EnrollmentLimit

After some further discussions, you have come up with some additional business rules to help you to create the initial design:

- An instructor teaches one, none, or many sections of a course in a given semester.
- An instructor specifies preferred time slots.
- Scheduling data is kept for each semester, uniquely identified by semester and year.
- A room can be scheduled for one section or no section during one time slot in a given semester of a given year. However, one room can participate in many schedules, one schedule, or no schedules; one time slot can participate in many schedules, one schedule, or no schedules; one section can participate in many schedules, one schedule, or no schedules. Hint: Can you associate this with anything that you have seen before?
- A room can have one type of media, several types of media, or no media.
- Instructors are trained to use one, none, or many types of media.
- A lab has one or more computer types. However, a classroom does not have any computers.
- A room cannot be both a classroom and a lab. There also are no other room types to be incorporated in the system.

16. Draw a class diagram, showing the relevant classes, attributes, operations, and relationships, for the following situation: Wally Los Gatos and his partner Henry Chordate have formed a new limited partnership, Fin and Finicky Security Consultants. Fin and Finicky consults with corporations to determine their security needs. You have been hired by Wally and Henry to design a database management system to help them manage their business.

    Due to a recent increase in business, Fin and Finicky has decided to automate their client tracking system. You and your team have done a preliminary analysis and come up with the following set of classes, attributes and business rules:

    Consultant

    There are two types of consultants: business consultants and technical consultants. Business consultants are contacted by a business in order to first determine security needs and provide an estimate for the actual services to be performed.

    Technical consultants perform services according to the specifications developed by the business consultants.

    Attributes of business consultant are the following: EmployeeID (identifier), Name, Address (Street, City, State, Zip code), Telephone, DateofBirth, Age, Business Experience (Number of years, Type of business (or businesses)), Degrees Received

    Attributes of technical consultant are the following: EmployeeID (identifier), Name, Address (Street, City, State, Zip code), Telephone, DateofBirth, Age, Technical Skills, Degrees Received

    Customer

    Customers are businesses that have asked for consulting services. Attributes of customer are: CustomerID (identifier), Company name, Address (Street, City, State, Zip code), ContactName, ContactTitle, ContactTelephone, BusinessType, NumberOfEmployees

    Location

    Customers can have multiple locations. Attributes of location are: CustomerID, LocationID (which is unique only for each CustomerID), Address (Street, City, State, Zip code), Telephone, BuildingSize

    Service

    A security service is performed for a customer at one or more locations. Before services are performed, an estimate is prepared. Attributes of service are: ServiceID (identifier), Description, Cost, Coverage, ClearanceRequired

    Additional Business Rules

    In addition to the classes outlined above, the following information will need to be stored and should be shown in the model. These may be classes but they also reflect a relationship between more than one class.

    Estimates: Date, Amount, Business consultant, Services, Customer

    Services Performed: Date, Amount, Technical Consultant, Services, Customer

    In order to construct the class diagram, you may assume the following: A customer can have many consultants providing many services. We wish to track both actual services performed as well as services offered. Therefore, there should be two relationships between customer, service and consultant, one to show services performed and one to show services offered as part of the estimate.

17. In Chapter 11, we presented a case study for the Fitchwood Insurance Agency. As you may recall, we developed the ER Diagram shown in Figure 14-25 for this agency:

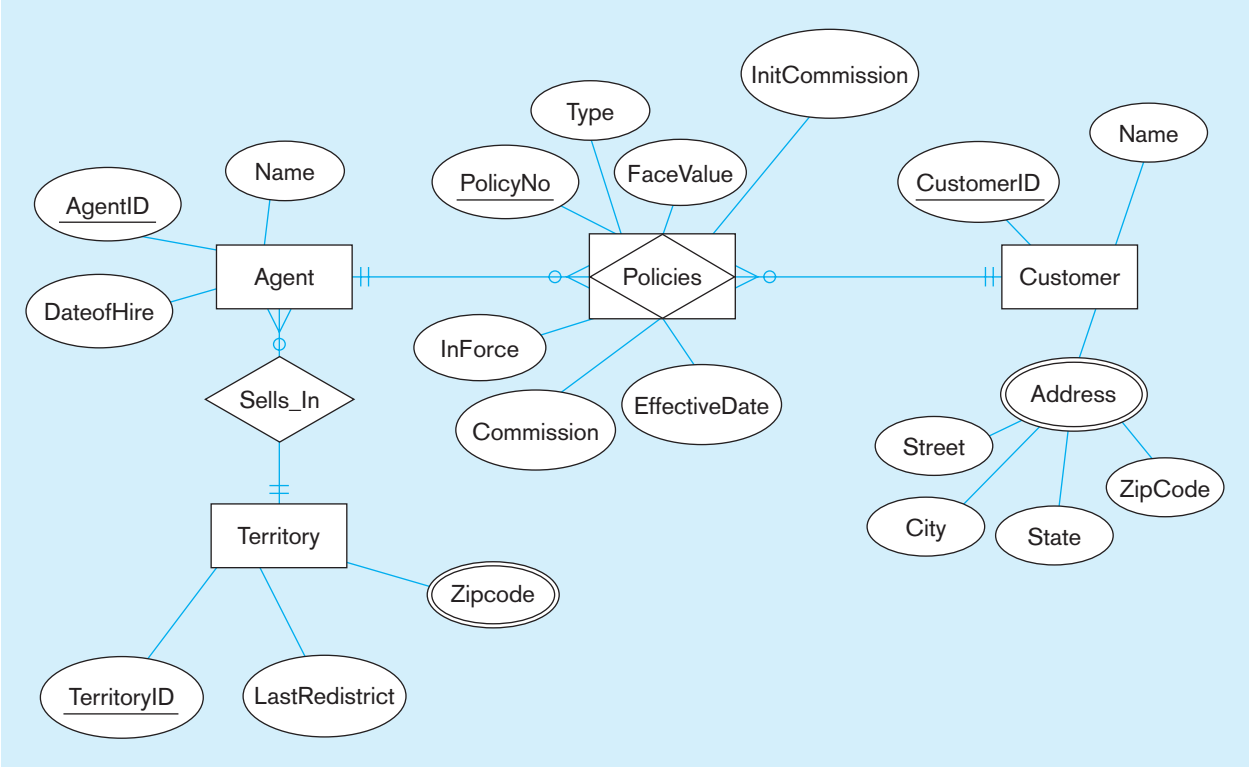


Figure 14-25

Please convert this ER Diagram into a class diagram. Please state any assumptions that you make.

18. Assume that at Pine Valley Furniture Company each product (described by product number, description, and cost) comprises at least three components (described by component number, description, and unit of measure), and components are used to make one or many products. In addition, assume that components are used to make other components and that raw materials are also considered to be components. In both cases of components, we need to keep track of how many components go into making something else. Draw a class diagram for this situation; indicate the multiplicities for all the relationships you identified in the diagram.
19. Pine Valley Furniture Company has implemented electronic payment methods for some customers. These customers will no longer require an invoice. The SendInvoice and ReceivePayment methods will still be used for those customers who always pay by cash or check. However, a new method is needed to receive an electronic payment from those customers who use the new payment method. How will this change impact the Pine Valley Furniture class diagram? Please redraw the diagram to include any changes that you feel are necessary.
20. Within the Pine Valley Furniture class diagram, is there a need to add any derived associations or derived relationships? If so, please redraw the diagram to represent this.

Field Exercises

1. Interview a friend or family member to elicit from them common examples of superclass/subclass relationships. You will have to explain the meaning of this term and provide a common example, such as: PROPERTY: RESIDENTIAL, COMMERCIAL; or BONDS: CORPORATE, MUNICIPAL. Use the information they provide to construct a class diagram segment and present it to this person. Revise if necessary until it seems appropriate to you and your friend or family member.
2. Visit two local small businesses, one in the service sector and one in manufacturing. Interview employees from these organizations to obtain examples of both superclass/subclass relationships and operational business rules (such as "A customer can return merchandise only if the customer has a valid sales slip"). In which of these environments is it easier to find examples of these constructs? Why?
3. Ask a database administrator or database or systems analyst in a local company to show you an EER (or E-R) diagram for

one of the organization's primary databases. Translate this diagram into a class diagram.

4. Interview a systems analyst in a local company who uses object-oriented programming and system development tools. Ask to see any analysis and design diagrams the analyst has drawn of the database and applications. Compare these diagrams to the ones in this chapter. What differences do you see? What additional features and notations are used and what are their purpose?

References

Booch, G. 1994. *Object-Oriented Analysis and Design with Applications*, 2nd ed. Redwood City, CA: Benjamin/Cummings.

Coad, P., and E. Yourdon. 1991. *Object-Oriented Design*. Upper Saddle River, NJ: Prentice Hall.

Fowler, M. 2000. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*, 2d ed. Reading, MA: Addison Wesley Longman.

George, J., D. Batra, J. Valacich, and J. Hoffer. 2005. *Object Oriented Systems Analysis and Design*. Upper Saddle River, NJ: Prentice Hall.

Hoffer, J., J. George, and J. Valacich. 2005. *Modern Systems Analysis and Design*. Upper Saddle River, NJ: Prentice Hall.

Jacobson, I., M. Christerson, P. Jonsson, and G. Overgaard. 1992. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Reading, MA: Addison-Wesley.

Quatrani, T. 2000. *Visual Modeling with Rational Rose 2000 and UML*. Upper Saddle River, NJ: Addison-Wesley.

Rumbaugh, J., M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. 1991. *Object-Oriented Modeling and Design*. Upper Saddle River, NJ: Prentice Hall.

UML Document Set. 1997. Version 1.0 (January). Santa Clara, CA: Rational Software Corp.

UML Notation Guide. 1997. Version 1.0 (January). Santa Clara, CA: Rational Software Corp.

Further Reading

Eriksson, H., and M. Penker. 1998. *UML Toolkit*. New York: Wiley.

Linthicum, D. 1996. "Objects Meet Data." *DBMS* 9,9 (September): 72–78. See also Appendix D.

Web Resources

www.omg.org This is the Website of the Object Management Group, a leading industry association concerned with object-oriented analysis and design.