Appendix A

Object-Oriented Analysis and Design*

After studying this appendix, you should be able to:

- Define the following key terms: *association, class diagram, event, object, object class, operation, sequence diagram, state, state transition, Unified Modeling Language,* and *use case.*
- Describe the concepts and principles underlying the object-oriented approach.
- Develop a simple requirements model using use-case diagrams.
- Develop a simple object model using class diagrams.
- Develop simple requirements models using state and sequence diagrams.

The Object-Oriented Modeling Approach

In Chapters 1 through 10, you learned about traditional methods of systems analysis and design. You also learned how to use data-flow diagrams and entity-relationship diagrams to model your system. In some environments, an object-oriented rather than a traditional approach is needed. This appendix covers the techniques and graphical diagrams that systems analysts use for object-oriented analysis and design. As with the traditional modeling techniques, the deliverables from project activities using object-oriented modeling are data-flow and entity-relationship diagrams and repository descriptions. A major characteristic of these diagrams in object-oriented modeling is how tightly they are linked with each other. The object-oriented modeling approach provides several benefits, including:

1. The ability to tackle more challenging problem domains
2. Improved communication among users, analysts, designers, and programmers
3. Reusability of analysis, design, and programming results
4. Increased consistency among the models developed during object-oriented analysis, design, and programming

An object-oriented systems development life cycle consists of a progressively developing representation of a system component (what we will call an *object*) through the phases of analysis, design, and implementation. In the early stages of development, the model built is abstract, focusing on external qualities of the application system such as data structures, timing and sequence of processing operations, and how users interact with the system. As the model evolves, it becomes more and more detailed, shifting the focus to how the system will be built and how it should function.

In the analysis phase, a model of the real-world application is developed showing its important properties. It abstracts concepts from the application

*The original version of this appendix was written by Professor Atish P. Sinha.

domain and describes what the intended system must do, rather than how it will be done. The model specifies the functional behavior of the system, independent of concerns relating to the environment in which it is to be finally implemented. In the design phase, the application-oriented analysis model is adapted and refined to suit the target implementation environment. That is followed by the implementation phase, where the design is implemented using a programming language and/or a database management system. The techniques and notations that are incorporated into a standard object-oriented language are called the **Unified Modeling Language (UML).**

Unified Modeling Language (UML)
A notation that allows the modeler to specify, visualize, and construct the artifacts of software systems, as well as business models.

The techniques and notations within UML include:

- Use cases, which represent the functional requirements or the "what" of the system
- Class diagrams, which show the static structure of data and the operations that act on the data
- State diagrams, which represent dynamic models of how objects change their states in response to events
- Sequence diagrams, which represent dynamic models of interactions between objects

The Unified Modeling Language (UML) allows the modeler to specify, visualize, and construct the artifacts of software systems, as well as business models. It builds upon and unifies the semantics and notations of leading object-oriented methods and has been adopted as an industry standard.

The UML notation is useful for graphically depicting object-oriented analysis and design models. 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 to communicate with domain experts, users, and other stakeholders. To represent a complex system effectively, the model developed needs to have a small set of independent views of the system. UML allows you to represent multiple views of a system using a variety of graphical diagrams, such as the use-case diagram, class diagram, state diagram, sequence diagram, and collaboration diagram. The underlying model integrates those views so that the system can be analyzed, designed, and implemented in a complete and consistent fashion.

We first show how to develop a use-case model during the requirements analysis phase. Next, we show how to model the static structure of the system using class and object diagrams. You then learn how to capture the dynamic aspects using state and sequence diagrams. Finally, we provide a brief description of component diagrams, which are generated during the design and implementation phases.

Use-Case Modeling

Use-case modeling is applied to analyze the functional requirements of a system. Use-case modeling is done in the early stages of system development (during the analysis phase) to help developers understand the functional requirements of the system without worrying about how those requirements will be implemented. The process is inherently iterative; developers need to involve the users in discussions throughout the model development process and finally come to an agreement on the requirements specification.

Actor
An external entity that interacts with the system (similar to an external entity in data-flow diagramming).

Use case
A sequence of related actions initiated by an actor; it represents a specific way to use the system.

A use-case model consists of actors and use cases. An **actor** is an external entity that interacts with the system (similar to an external entity in data-flow diagramming). It is someone or something that exchanges information with the system. A **use case** represents a sequence of related actions initiated by an

actor; it is a specific way of using the system. An actor represents a role that a user can play. The actor's name should indicate that role. Actors help you to identify the use cases they carry out.

During the requirements analysis stage, the analyst sits down with the intended users of the system and makes a thorough analysis of what functions they desire from the system. These functions are represented as use cases. For example, a university registration system has a use case for class registration and another for student billing. These use cases, then, represent the typical interactions the system has with its users.

In UML, a use-case model is depicted diagrammatically, as in Figure A-1. This **use-case diagram** is for a university registration system, which is shown as a box. Outside the box are four actors—Student, Registration clerk, Instructor, and Bursar's office—that interact with the system (shown by the lines touching the actors). An actor is shown using a stick figure with its name below. Inside the box are four use cases—*Class registration*, *Registration for special class*, *Prereq courses not completed*, and *Student billing*—which are shown as ellipses with their names inside. These use cases are performed by the actors outside the system.

Use-case diagram
A diagram that depicts the use cases and actors for a system.

A use case is always initiated by an actor. For example, *Student billing* is initiated by the Bursar's office. A use case can interact with actors other than the one that initiated it. The *Student billing* use case, although initiated by the Bursar's office, interacts with the Students by mailing them tuition invoices. Another use case, *Class registration*, is carried out by two actors, Student and Registration clerk. This use case performs a series of related actions aimed at registering a student for a class.

The numbers on each end of the interaction lines indicate the number of instances of the use case with which the actor is associated. For example, the Bursar's office causes many (*) *Student billing* use-case instances to occur, each one for exactly one student.

A use case represents a complete functionality. You should not represent an individual action that is part of an overall function as a use case. For example, although submitting a registration form and paying tuition are two actions performed by users (students) in the university registration system, we do not show them as use cases, because they do not specify a complete course of events; each of these actions is executed only as part of an overall function or

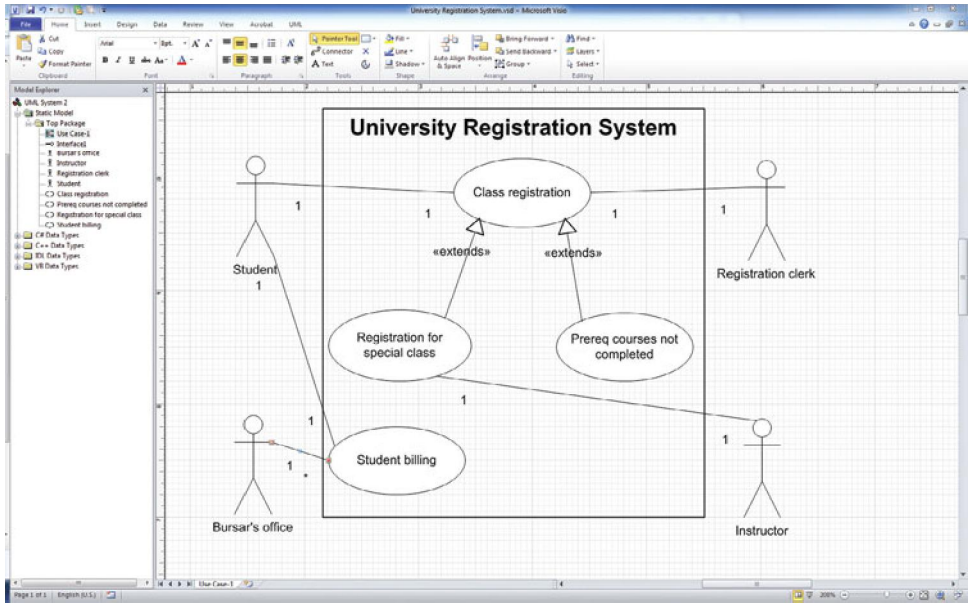

FIGURE A-1
Use-case diagram for a university registration system drawn using Microsoft Visio.

use case. You can think of "Submit registration form" as one of the actions of the *Class registration* use case, and "Pay tuition" as one of the actions of the *Student billing* use case.

A use case may participate in relationships with other use cases. An *extends relationship*, shown in Microsoft Visio as a line with a hollow triangle pointing toward the extended use case and labeled with the "<<extends>>" symbol, extends a use case by adding new behaviors or actions. In Figure A-1, for example, the *Registration for special class* use case extends the *Class registration* use case by capturing the additional actions that need to be performed in registering a student for a special class. Registering for a special class requires prior permission of the instructor, in addition to the other steps carried out for a regular registration. You may think of *Class registration* as the basic course, which is always performed—independent of whether the extension is performed or not—and *Registration for special class* as an alternative course, which is performed only under special circumstances.

Another example of an extends relationship is that between the *Prereq courses not completed* and *Class registration* use cases. The former extends the latter in situations where a student registering for a class has not taken the prerequisite courses.

Figure A-2 shows a use-case diagram for Hoosier Burger. The Customer actor initiates the *Order food* use case; the other actor involved is the Service Person. A specific scenario would represent a customer placing an order with a service person.

So far you have seen one kind of relationship, extends, between use cases. Another kind of relationship is *included*, which arises when one use case references another use case. An include relationship is also shown diagrammatically as a dashed line with a hollow arrowhead pointing toward the use case that is being used; the line is labeled with the "<<include>>" symbol. In Figure A-2, for example, the include relationship between the *Reorder supplies* and *Track sales and inventory data* use cases implies that the former uses the latter while

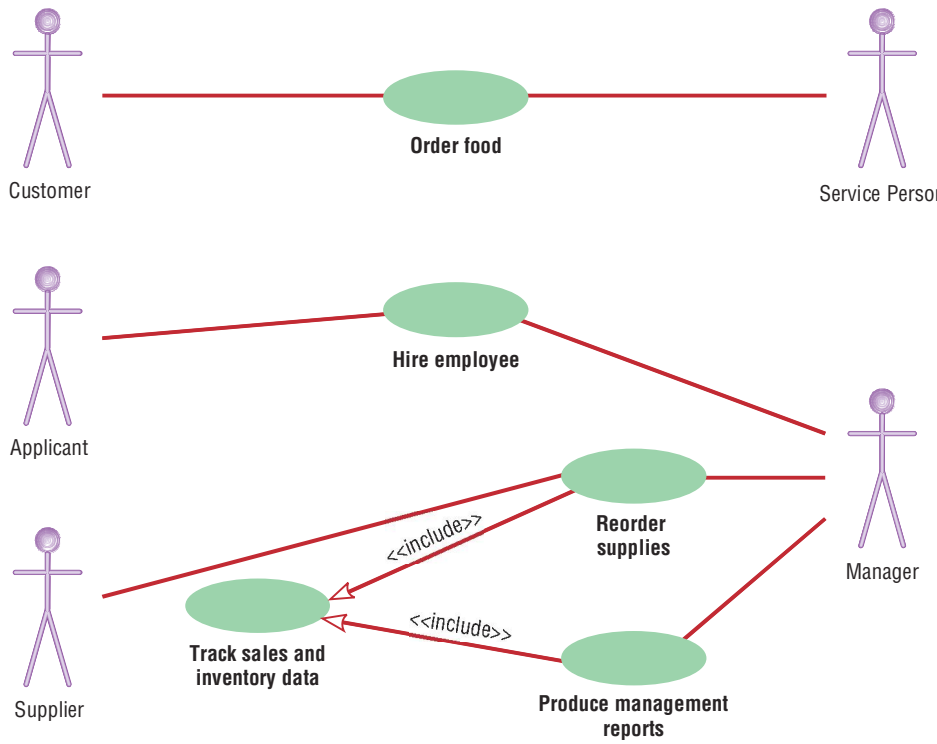

FIGURE A-2
Use-case diagram for a Hoosier Burger system.

executing. Simply put, when a manager reorders supplies, the sales and inventory data are tracked. The same data are also tracked when management reports are produced, so there is another include relationship between the *Produce management reports* and *Track sales and inventory data* use cases.

The *Track sales and inventory data* is a generalized use case, representing the common behavior among the specialized use cases, *Reorder supplies* and *Produce management reports*. When *Reorder supplies* or *Produce management reports* is performed, the entire *Track sales and inventory data* is used.

Object Modeling: Class Diagrams

In the object-oriented approach, we model the world in objects. An **object** is an entity that has a well-defined role in the application domain and has state, behavior, and identity. An object is a concept, abstraction, or thing that makes sense in an application context. An object 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.).

Object
An entity that has a well-defined role in the application domain and has state, behavior, and identity.

An object has a state and 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, its **behavior** represents how an object acts and reacts. 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.

State
A condition that encompasses an object's properties (attributes and relationships) and the values those properties have.

Behavior
Represents how an object acts and reacts.

Consider the example of a student, Mary Jones, represented as an object. 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 an identity, that is, no two objects are the same. For example, if two Student instances have the same name and date of birth, they are essentially two different objects. Even if those two instances have identical values for all the attributes, 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 an **object class** (a set of objects that shares a common structure and a common behavior) graphically in a class diagram as in Figure A-3A. 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.

Object class
A set of objects that shares a common structure and a common behavior.

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

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*.

An **object diagram,** also known as an *instance diagram*, is a graph of instances that are compatible with a given class diagram. In Figure A-3B, we have shown an object diagram with two instances, one for each of the two

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

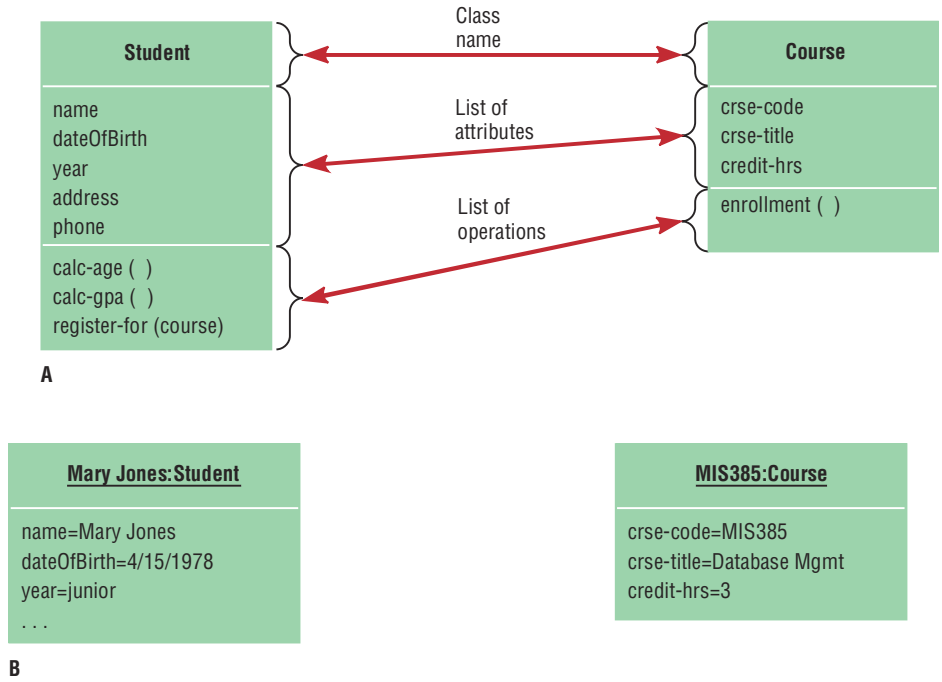

FIGURE A-3
UML class and object diagrams: (A) Class diagram showing two classes, (B) Object diagram with two instances.

classes that appear in Figure A-3A. A static object diagram 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, who belongs to the Student class. The values of the name, dateOfBirth, and year attributes are also shown.

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

An **operation,** such as calc-gpa in Student (see Figure A-3A), 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.*

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

Representing Associations

An **association** is a relationship among 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), as shown in Figure A-4. An association is depicted as a solid line between the participating classes. The end of an association where it connects to a class is called an **association role.** A role may be explicitly named with a label near the end of an association (see the "manager" role in Figure A-4A). The role name indicates the role played by the class attached to the end near which the name appears. For example, the manager role at one end of the Manages relationship implies that an employee can play the role of a manager.

Association
A relationship among object classes.

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

Each role has a **multiplicity,** which indicates how many objects participate in a given association relationship. In a class diagram, a multiplicity specification is

Multiplicity
An indication of how many objects participate in a given relationship.

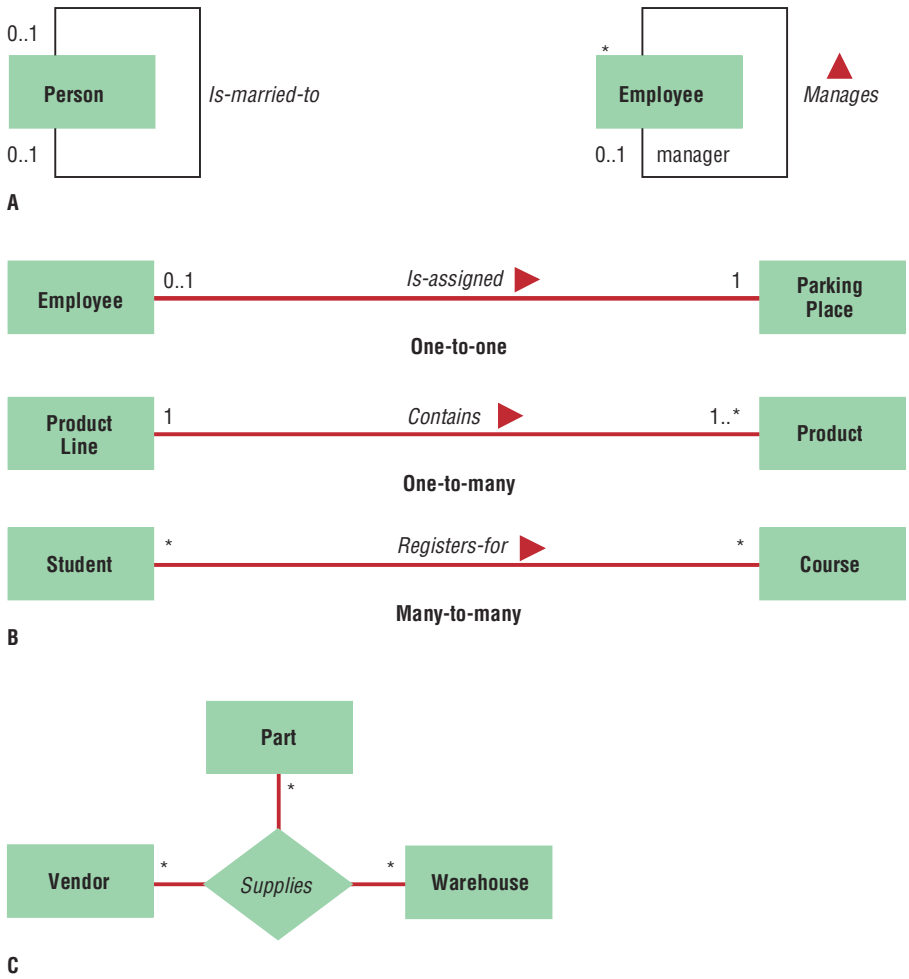

FIGURE A-4
Examples of association relationships of different degrees: (A) Unary, (B) Binary, (C) Ternary.

shown as a text string representing an interval (or intervals) of integers in the following format: lower bound..upper bound. The interval is considered to be closed, which means that the range includes both the lower and upper bounds. 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 0 and a maximum of 1 (optional one), whereas * (or equivalently, 0..*) represents the range from 0 to infinity (optional many). A single 1 stands for 1..1, implying that exactly one object participates in the relationship (mandatory one).

Figure A-4B shows three binary relationships: *Is-assigned* (one-to-one), *Contains* (one-to-many), and *Registers-for* (many-to-many). A solid triangle next to an association name shows the direction in which the association is read. For example, the *Contains* association is read from Product Line to Product.

Figure A-4C shows a ternary relationship called *Supplies* among Vendor, Part, and Warehouse. As in an E-R diagram, we represent a ternary relationship using a diamond symbol and place the name of the relationship there.

The class diagram in Figure A-5A (known as a *static structure chart* in Microsoft Visio) shows binary associations between Student and Faculty,

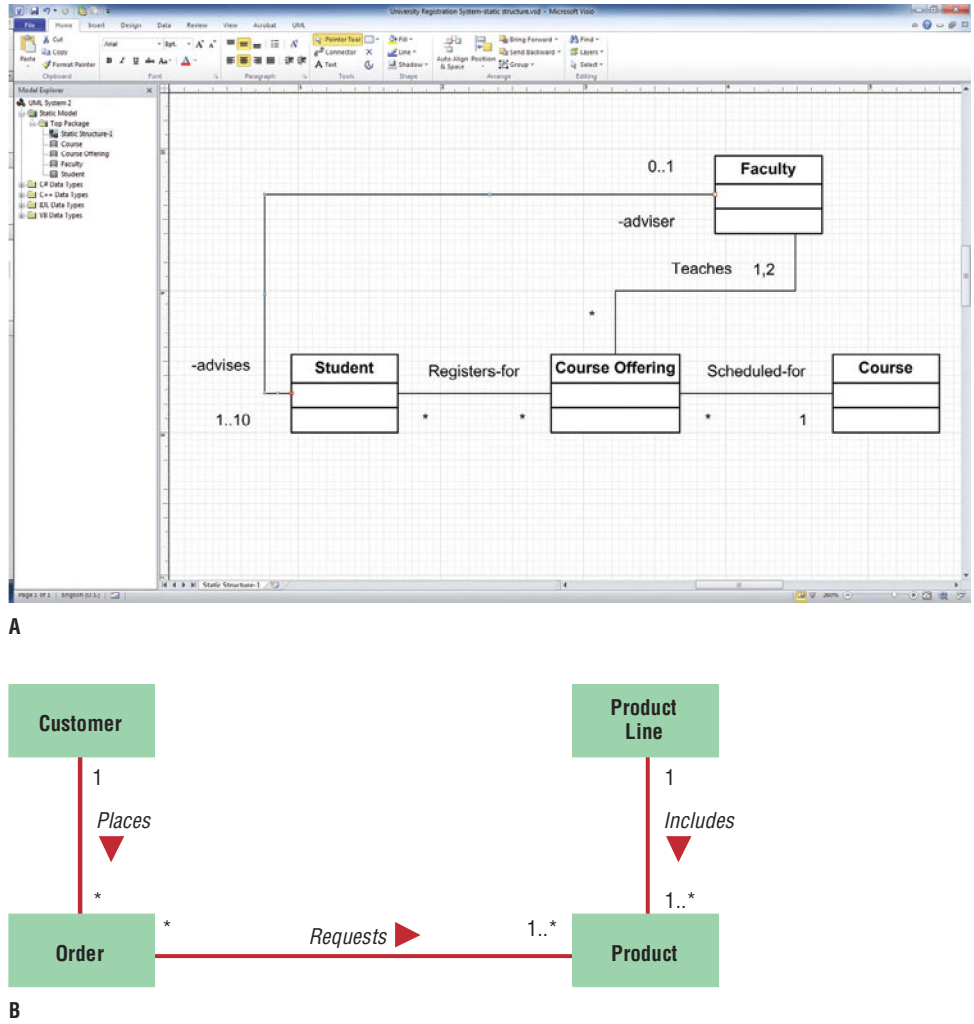

FIGURE A-5
Examples of binary association relationships: (A) University example (a static structure chart in Microsoft Visio), (B) Customer order example.

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 adviser, 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. Notice that a faculty member can play two roles: instructor and adviser. Figure A-5B shows another example of a class diagram, that for a customer order, using standard UML notation.

Representing Generalization

In the object-oriented approach, you can abstract the common features (attributes and operations) among multiple classes, as well as the relationships they participate in, into a more general class. This process is known as *generalization*. The classes that are generalized are called *subclasses*, and the class they are generalized into is called a *superclass*.

Consider the example shown in Figure A-6A. Here, the three types of employees are hourly employees, salaried employees, and consultants. The features that are shared by all employees—empName, empNumber, address, dateHired, and printLabel—are stored in the Employee superclass, whereas the

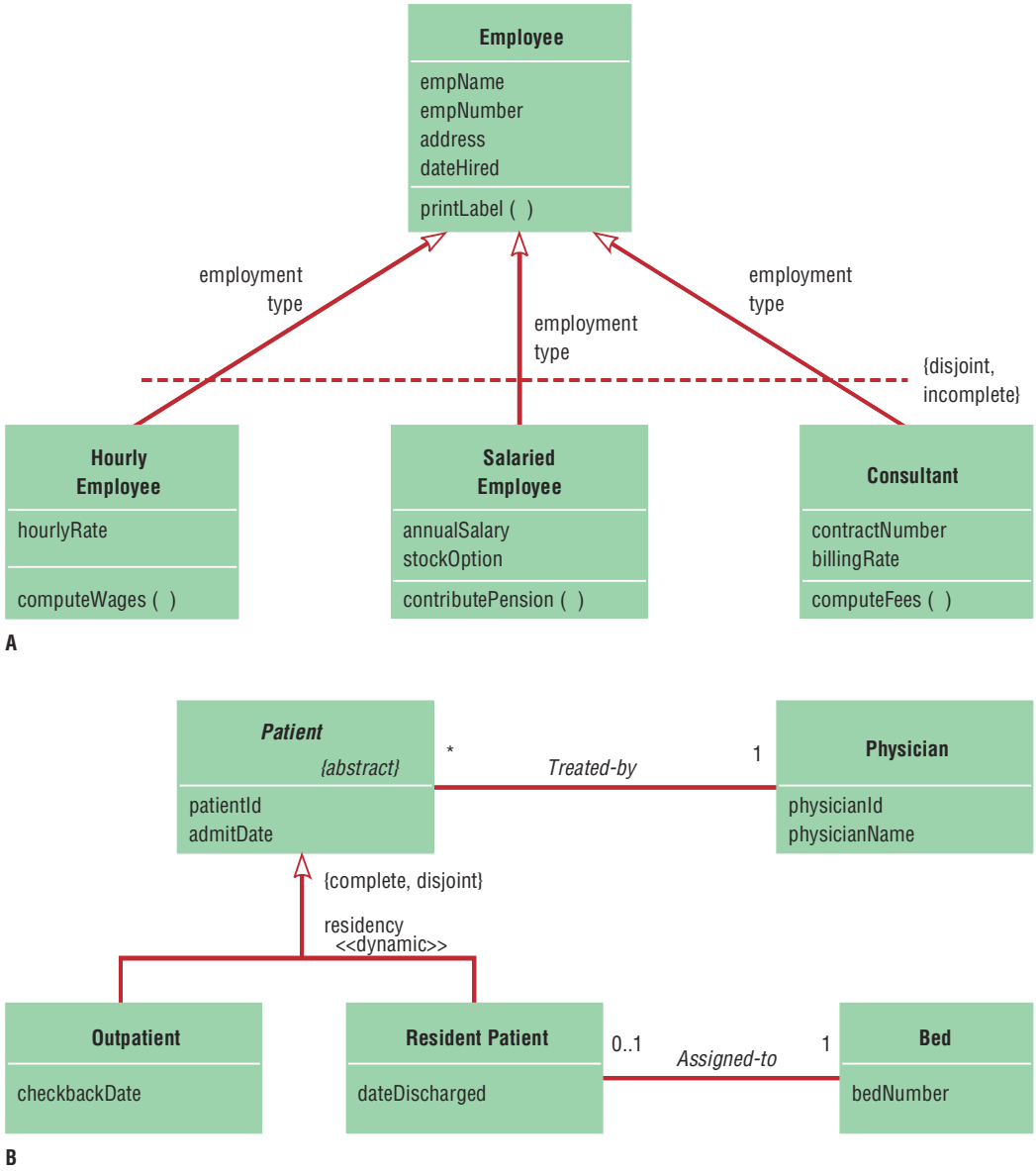

FIGURE A-6
Examples of generalization, inheritance, and constraints: (A) Employee superclass with three subclasses, (B) Abstract patient class with two concrete subclasses.

features 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 arrowhead 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 arrowhead pointing toward the superclass. In Figure A-6B, for instance, we have combined the generalization paths from Outpatient to Patient, and from Resident Patient to Patient, into a shared segment with an arrowhead pointing toward Patient.

You can indicate the basis of a generalization by specifying a discriminator next to the path. A discriminator shows which property of an object class is being abstracted by a particular generalization relationship. For example, in Figure A-6A, we discriminate the Employee class on the basis of employment type (hourly, salaried, consultant).

A subclass inherits all the features from its superclass. For example, 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.

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 writes only code that is unique to the new, refined subclass of an existing class. Proponents of the object-oriented model claim that code reuse results in productivity gains of several orders of magnitude.

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.

Notice that in Figure A-6B 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. A class that can have direct instances (e.g., Outpatient or Resident Patient) is called a **concrete class.**

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. Semantic constraints among the subclasses can be expressed using the *complete*, *incomplete*, *disjoint*, and *overlapping* keywords. *Complete* means that every instance must be an instance of some subclass, whereas *incomplete* means that an instance may be an instance of the superclass only. *Disjoint* means that no instance can be an instance of more than one subclass at the same time, whereas *overlapping* allows concurrent participation in multiple subclasses.

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 an association relationship (with the added "part-of" semantics) and is represented with a hollow diamond at the aggregate end. For example, Figure A-7 shows a personal computer as an aggregate of CPU (up to four for multiprocessors), hard disk, monitor, keyboard, and other objects. 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) CPU objects, one of its parts. In contrast, generalization relates to object classes: An object (e.g., Mary Jones) is simultaneously an instance of its class (e.g., Graduate Student) and its superclass (e.g., Student).

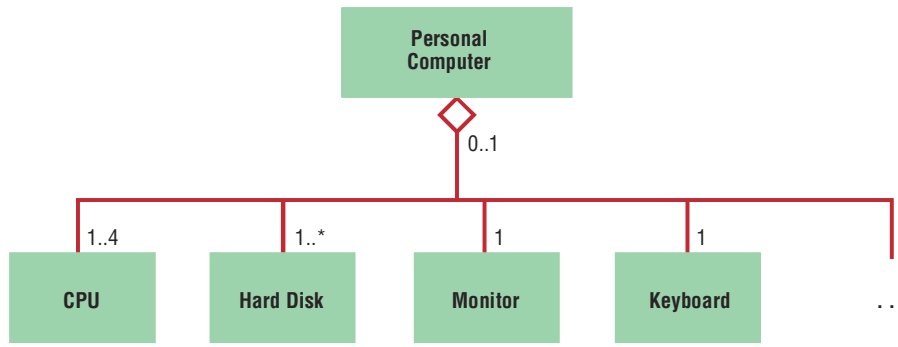

FIGURE A-7
Example of aggregation.

Dynamic Modeling: State Diagrams

In this section, we show you how to model the dynamic aspects of a system from the perspective of state transitions. In UML, state transitions are shown using state diagrams. A state diagram depicts the various state transitions or changes an object can experience during its lifetime, along with the events that cause those transitions.

A *state* is a condition during the life of an object during which it satisfies some condition(s), performs some action(s), or waits for some event(s). The state changes when the object receives some event; the object is said to undergo a **state transition.** The state of an object depends on its attribute values and links to other objects.

State transition
The changes in the attributes of an object or in the links an object has with other objects.

An **event** is something that takes place at a certain point in time. It is a noteworthy occurrence that triggers a state transition. Some examples of events are: a customer places an order, a student registers for a class, a person applies for a loan, and a company hires a new employee. For the purpose of modeling, an event is considered to be instantaneous, though, in reality, it might take some time. A state, on the other hand, spans a period of time. An object remains in a particular state for some time before transitioning to another state. For example, an Employee object might be in the Part-time state (as specified in its employment-status attribute) for a few months, before transitioning to a Full-time state, based on a recommendation from the manager (an event).

Event
Something that takes place at a certain point in time; it is a noteworthy occurrence that triggers a state transition.

In UML, a state is shown as a rectangle with rounded corners. In Figure A-8, for example, we have shown different states of a Student object, such as Inquiry, Applied, Approved, Rejected, and so on. This state diagram shows how the object transitions from an initial state (shown as a small, solid, filled circle) to other states when certain events occur or when certain conditions are satisfied. When a new Student object is created, it is in its initial state. The event that created the object, inquires, and transitions it from the initial state to the Inquiry state. When a student in the Inquiry state submits an application for admission,

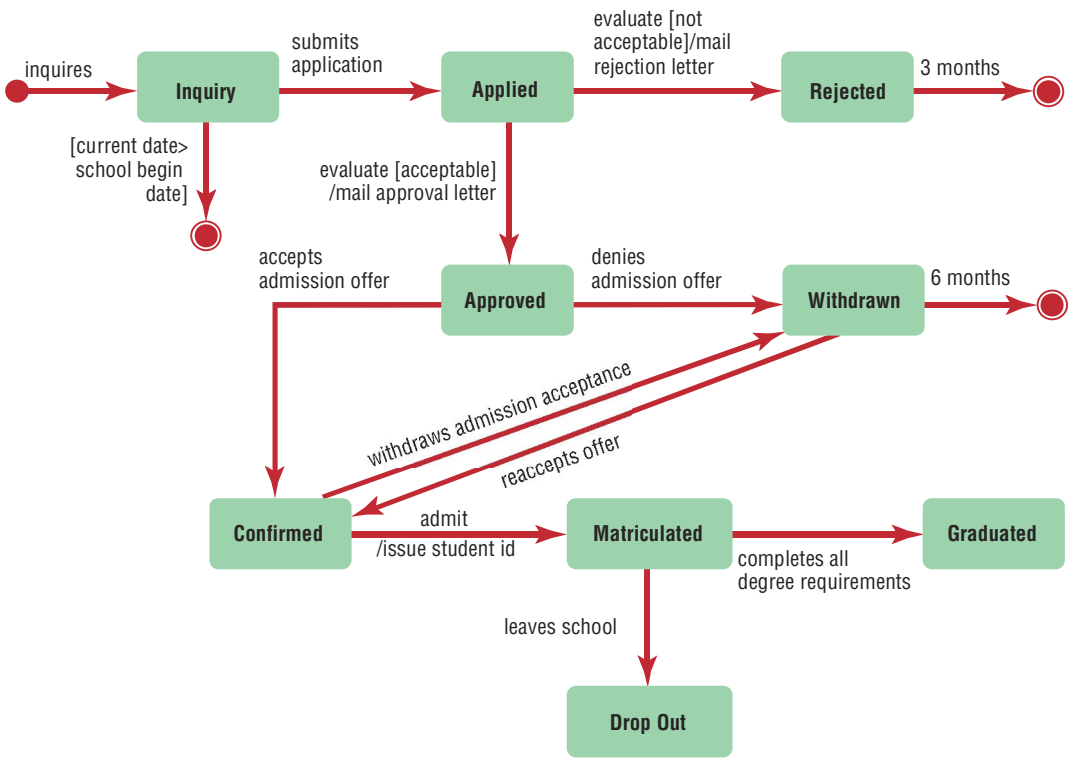

FIGURE A-8
State diagram for the Student object.

the object transitions to the Applied state. The transition is shown as a solid arrow from Inquiry (the source state) to Applied (the target state), labeled with the name of the event, Submits application.

A transition may be labeled with a string consisting of the event name, parameters of the event, guard condition, and action expression. A transition, however, does not have to show all the elements; it shows only those relevant to the transition. For example, we label the transition from Inquiry to Applied with simply the event name. But, for the transition from Applied to Approved, we show the event name (evaluate), the guard condition (acceptable), and the action taken by the transition (mail approval letter). It simply means that an applicant is approved for admission if the admissions office evaluates the application and finds it acceptable. If acceptable, a letter of approval is mailed to the student. The guard condition is shown within square brackets, and the action is specified after the "/" symbol.

If the evaluate event results in a not-acceptable decision (another guard condition), a rejection letter is mailed (an action), and the Student object undergoes a state transition from the Applied to the Rejected state. It remains in that state for three months before reaching the final state. In the diagram, we have shown an elapsed-time event, three months, indicating the amount of time the object waits in the current state before transitioning. The final state is shown as a bull's eye: a small, solid, filled circle surrounded by another circle. After transitioning to the final state, the Student object ceases to exist.

Notice that the Student object may transition to the final state from two other states: Inquiry and Withdrawn. For the transition from Inquiry, we have not specified any event name or action, but we have shown a guard condition, current date > school begin date. This condition implies that the Student object ceases to exist beyond the first day of school unless, of course, the object has moved in the meantime from the Inquiry state to some other state.

The state diagram shown in Figure A-8 captures all the possible states of a Student object, the state transitions, the events or conditions that trigger those transitions, and the associated actions. For a typical student, it captures the student's sojourn through college, right from the time when he or she expressed an interest in the college until graduation.

Dynamic Modeling: Sequence Diagrams

In this section we show how to design some of the use cases we identified earlier in the analysis phase. A use-case design describes how each use case is performed by a set of communicating objects. In UML, an interaction diagram is used to show the pattern of interactions among objects for a particular use case. The two types of interaction diagrams are sequence diagrams and collaboration diagrams. We show you how to design use cases using sequence diagrams.

Sequence diagram
A depiction of the interactions among objects during a certain period of time.

A **sequence diagram** depicts the interactions among objects during a certain period of time. Because the pattern of interactions varies from one use case to another, each sequence diagram shows only the interactions pertinent to a specific use case. It shows the participating objects by their lifelines and the interactions among those objects—arranged in time sequence—by the messages they exchange with one another.

Figure A-9 shows a sequence diagram for a scenario, discussed in the next section, of the Class registration use case in which a student registers for a course that requires one or more prerequisite courses. The vertical axis of the diagram represents time, and the horizontal axis represents the various participating objects. Time increases as we go down the vertical axis. The diagram has six objects, from an instance of Registration Window on the left, to an instance of Registration called *a New Registration* on the right. Each object is shown as

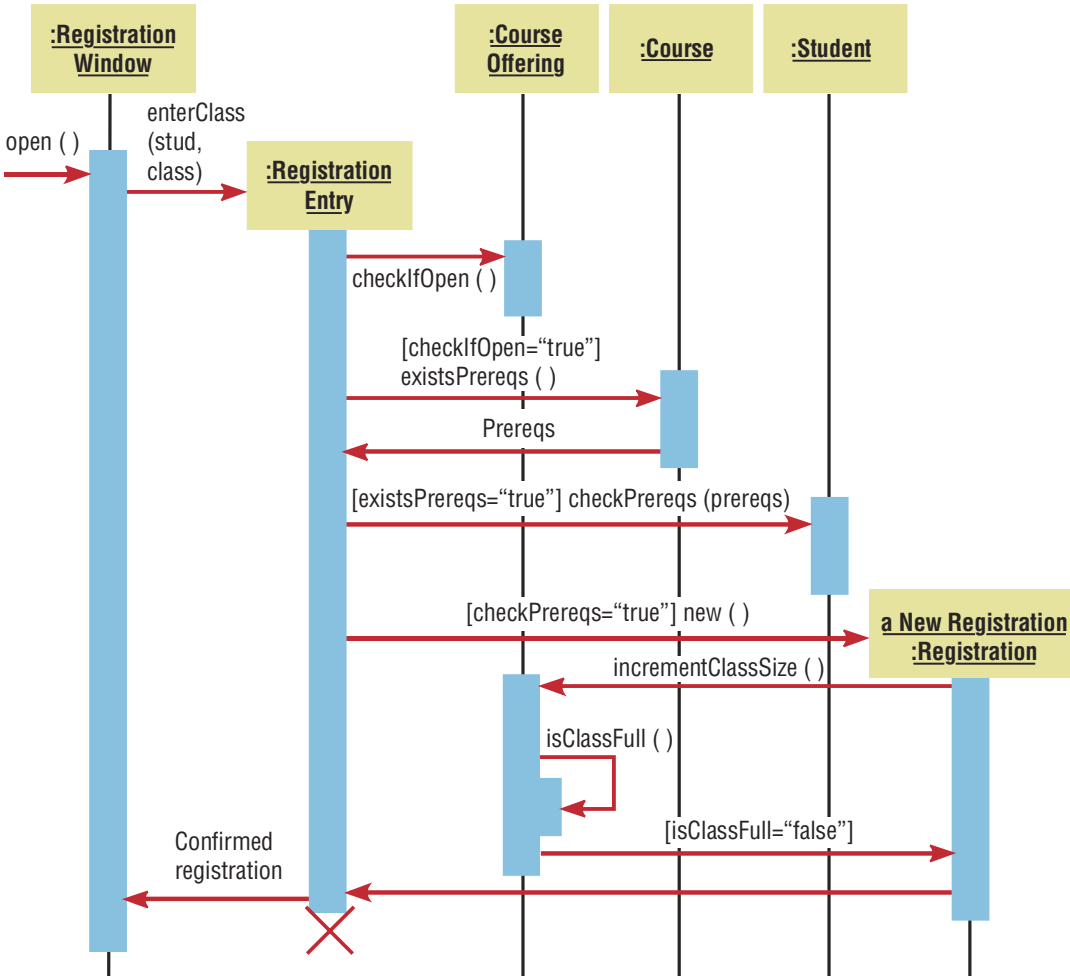

FIGURE A-9
Sequence diagram for a class registration scenario with prerequisites.

a vertical line called the *lifeline;* the lifeline represents the object's existence over a certain period of time. An object symbol—a box with the object's name underlined—is placed at the head of each lifeline.

A thin rectangle, superimposed on the lifeline of an object, represents an activation of the object. An **activation** shows the time period during which the object performs an operation, either directly or through a call to some subordinate operation. The top of the rectangle, which is at the tip of an incoming message, indicates the initiation of the activation, and the bottom, its completion.

Activation
The time period during which an object performs an operation.

Objects communicate with one another by sending messages. A message is shown as a solid arrow from the sending object to the receiving object. For example, the checkIfOpen message is represented by an arrow from the Registration Entry object to the Course Offering object.

There are different types of messages. Each type is indicated in a diagram by a particular type of arrowhead. A **synchronous message,** shown as a solid arrowhead, is one for which the caller has to wait for the receiving object to complete executing the called operation before it itself can resume execution. An example of a synchronous message is checkIfOpen. When a Registration Entry object sends this message to a Course Offering object, the latter responds by executing an operation called checkIfOpen (same name as the message). After the execution of this operation is completed, control is transferred back to the calling operation within Registration Entry with a return value of "true" or "false."

Synchronous message
A type of message in which the caller has to wait for the receiving object to finish executing the called operation before it can resume execution itself.

A synchronous message always has an associated return message. The message may provide the caller with some return value(s) or simply acknowledge to the caller that the operation called has been successfully completed. We have not shown the return for the checkIfOpen message; it is implicit. We have explicitly shown the return for the existsPrereqs message from Registration Entry to Course. The tail of the return message is aligned with the base of the activation rectangle for the existsPrereqs operation.

Simple message
A message that transfers control from the sender to the recipient without describing the details of the communication.

A **simple message** simply transfers control from the sender to the recipient without describing the details of the communication. As we have seen, the return of a synchronous message is a simple message. The "open" message in Figure A-9 is also a simple message; it simply transfers control to the Registration Window object.

Designing a Use Case with a Sequence Diagram

Let's see how we can design use cases. We will discuss the sequence diagram, shown in Figure A-9, for an instance of the *Class registration* use case, one in which the course has prerequisites. Here's a description of this scenario:

1. Registration Clerk opens the registration window and enters the registration information (student and class).
2. Check if the class is open.
3. If the class is open, check if the course has any prerequisites.
4. If the course has prerequisites, then check if the student has taken all of those prerequisites.
5. If the student has taken those prerequisites, then register the student for the class, and increment the class size by one.
6. Check if the class is full; if not, do nothing.
7. Display the confirmed registration in the registration window.

In response to the "open" message from Registration Clerk (external actor), the Registration Window pops up on the screen, and the registration information is entered. A new Registration Entry object is created, which then sends a checkIfOpen message to the Course Offering object (representing the class the student wants to register for). The two possible return values are true or false. In this scenario, the assumption is that the class is open. We have, therefore placed a guard condition, checkIfOpen = "true," on the message existsPrereqs. The guard condition ensures that the message will be sent only if the class is open. The return value is a list of prerequisites; the return is shown explicitly in the diagram.

For this scenario, the fact that the course has prerequisites is captured by the guard condition, existsPrereqs = "true." If this condition is satisfied, the Registration Entry object sends a checkPrereqs message, with "prereqs" as an argument, to the Student object to determine if the student has taken those prerequisites. If the student has taken all the prerequisites, the Registration Entry object creates an object called a *New Registration*, which denotes a new registration.

Next, a New Registration sends a message called incrementClassSize to Course Offering in order to increase the class size by one. The incrementClassSize operation within Course Offering then calls upon isClassFull, another operation within the same object. Assuming that the class is not full, the isClassFull operation returns control to the calling operation with a value of "false." Next, the incrementClassSize operation completes and relinquishes control to the calling operation within a New Registration.

Finally, on receipt of the return message from a New Registration, the Registration Entry object destroys itself (the destruction is shown with a large *X*) and

sends a confirmation of the registration to the Registration Window. Note that Registration Entry is not a persistent object; it is created on the fly to control the sequence of interactions and is deleted as soon as the registration is completed. In between, it calls several other operations within other objects by sequencing the following messages: checkIfOpen, existsPrereqs, checkPrereqs, and new.

Apart from the Registration Entry object, *a New Registration* is also created during the time period captured in the diagram. The messages that created these objects are represented by arrows pointing directly toward the object symbols. For example, the arrow representing the message called *new* is connected to the object symbol for a New Registration. The lifeline of such an object begins when the message that creates it is received (the vertical line is hidden behind the activation rectangle).

Moving to Design

When you move to design, you start with the existing set of analysis models and keep adding technical details. For example, you might add several interface classes to your class diagrams to model the windows that you will later implement using a GUI graphical user interface development tool such as Visual C# or Java. You would define all the operations in detail, specifying the procedures, signatures, and return values completely. If you decide to use a relational DBMS, you need to map the object classes and relationships to tables, primary keys, and foreign keys. The models generated during the design phase will therefore be much more detailed than the analysis models.

Figure A-10 shows a three-layered architecture, consisting of a User Interface package, a Business Objects package, and a Database package. The packages represent different generic subsystems of an information system. The dashed arrows represent the dependencies among the packages. For example, the User Interface package depends on the Business Objects package; the packages participate in a client-supplier relationship. If you make changes to some of the business objects, the interface (e.g., screens) might change.

A package consists of a group of classes. Classes within a package are cohesive. That is, they are tightly coupled. The packages themselves should be loosely coupled so that changes in one package do not affect the other packages a great deal. In the architecture of Figure A-10, the User Interface package contains all the windows, the Business Objects package contains the problem-domain objects that you identified during analysis, and the Database package contains a Persistence class for data storage and retrieval. In the university registration system that we considered earlier, the User Interface package could include Microsoft Windows class libraries for developing different types of windows. The Business Objects package would include all the domain classes, such as Student, Course, Course Offering, Registration, and so on. If you are using an SQL server, the classes in the Database package would contain operations for data storage, retrieval, and update (all using SQL commands).

During design, you would also refine the other analysis models. For example, you may need to show the interaction between a new window object you introduced during design and the other existing objects in a sequence diagram. Also, once you have selected a programming language for each of the operations shown in the sequence diagram, you should provide the exact names that you will be using in the program, along with the names of all the arguments.

In addition to the types of diagrams you have seen so far, two other types of diagrams—component diagrams and deployment diagrams—are pertinent during the design phase. A **component diagram** shows the software components or modules and their dependencies. For example, you can draw a component diagram showing the modules for source code, binary code, and executable

Component diagram
A diagram that shows the software components or modules and their dependencies.

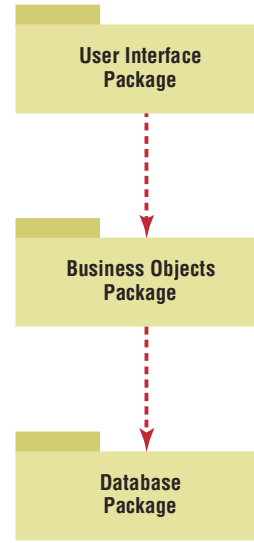

FIGURE A-10
An example of UML packages and dependencies.

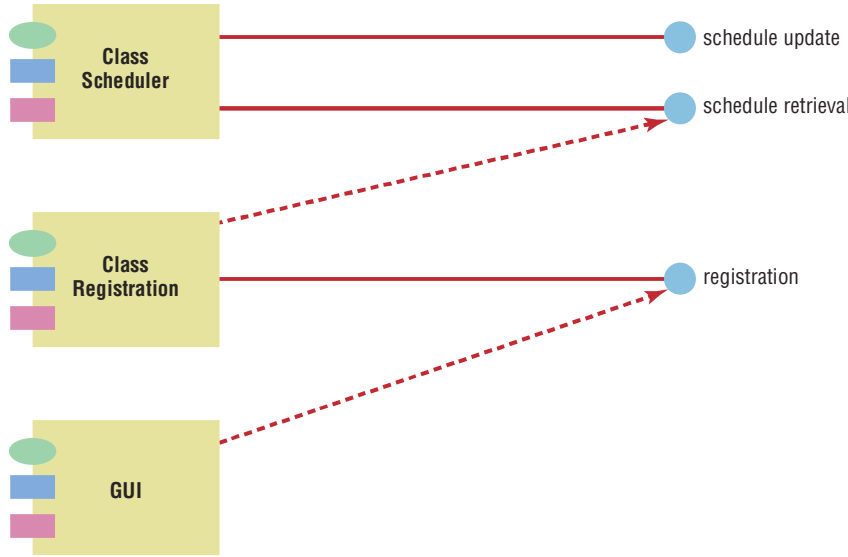

FIGURE A-11
A component diagram for class registration.

code and their dependency relationships. Figure A-11 shows a component diagram for the university registration system. In this figure, three software components have been identified: Class Scheduler, Class Registration, and GUI. The small circles in the diagram represent interfaces. The registration interface, for example, is used to register a student for a class, and the schedule update interface is used for updating a class schedule.

Another type of diagram, a deployment diagram (not illustrated), shows how the software components, processes, and objects are deployed into the physical architecture of the system. It shows the configuration of the hardware units (e.g., computers, communication devices) and how the software (components, objects, etc.) is distributed across the units. For example, a deployment diagram for the university registration system might show the topology of nodes in a client/server architecture and the deployment of the Class Registration component to a Windows NT Server and of the GUI component to client workstations.

When the design phase is complete, you move on to the implementation phase where you code the system. If you are using an object-oriented programming language, translating the design models to code should be relatively straightforward. Programming of the system is followed by testing. The system is developed after going through multiple iterations, with each new iteration providing a better version of the system. The models that you developed during analysis, design, and implementation are navigable in both directions.

Key Points Review

1. **Define the following key terms: *association, class diagram, event, object, object class, operation, sequence diagram, state, state transition, Unified Modeling Language,* and *use case.***

   An association is a relationship among object classes. 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. An event is something that takes place at a certain point in time; it is a noteworthy occurrence that triggers a state transition. An object is an entity that has a well-defined role in the application domain and has state, behavior, and identity; an object class is a set of objects that share a common structure and a common behavior. An operation is a function or a service that is provided by all the instances of a class. A sequence diagram depicts the interactions among objects during a certain period of time. A state encompasses an object's properties (attributes and relationships) and the values

those properties have; a state transition is the changes the attributes of an object or the links an object has with other objects. The Unified Modeling Language (UML) is a notation that allows the modeler to specify, visualize, and construct the artifacts of software systems, as well as business models. A use case is a complete sequence of related actions initiated by an actor; it represents a specific way to use the system.

2. **Describe the concepts and principles underlying the object-oriented approach.**

   The fundamental concept of the object-oriented approach is that we can model the world as a set of related objects with their associated states—attributes and behaviors. Through different uses of an object, the object's state changes. The internal implementation details of an object can be hidden from external view by the technique of encapsulation. A class of objects may be a superset or subset of other classes of objects, forming a generalization hierarchy or network of object classes. In this way an object may inherit the properties of the superclasses to which it is related. An object may also be a part of another more aggregate object.

3. **Develop a simple requirements model using use-case diagrams.**

   A use-case diagram consists of a set of related actions initiated by actors. A use case represents a complete functionality, not an individual action. A use case may extend another use case by adding new behaviors or actions. A use case may use another use case when one use case calls on another use case.

4. **Develop a simple object model using class diagrams.**

   A class diagram shows the static structure of object classes, their internal structure, and the relationships in which they participate. The structure of a class includes its name, attributes, and operations. Each object has an object identifier separate from its attributes. An object class can be either abstract (having no direct instances) or concrete (having direct instances). Object classes may have associations similar to relationships in the entity-relationship notation with multiplicity and degree. The end of an association where it connects to a class is labeled with an association role. A class diagram can also show the generalization relationships between object classes, and subclasses can be complete or incomplete and disjointed or overlapping. In addition, a class diagram may show the aggregation association among object classes.

5. **Develop simple requirements models using state and sequence diagrams.**

   State and sequence diagrams show the dynamic behavior of a system. A state diagram shows all the possible states of an object and the events that trigger an object to transition from one state to another. A state transition occurs by changes in the attributes of an object or in the links an object has with other objects. An object begins in an initial state and ends in a final state. A state may have a guard condition, which checks that certain object properties exist before the transition may occur. When a state transition occurs, specified actions may take place. A sequence diagram depicts the interactions among objects during a certain period of time. The vertical axis of the diagram represents time (going down the axis), and the horizontal axis represents the various participating objects. Each object has a lifeline, which represents the object's existence over a certain period. Objects communicate with one another by sending messages. Among the different types of messages are synchronous (for which the caller has to wait for the receiving object to complete the called operation before the caller can resume execution) and simple (for which control is transferred from the sender to the recipient).

Key Terms Checkpoint

Here are the key terms from the appendix. The page where each term is first explained is in parentheses after the term.

1. Abstract class (p. 370)
2. Activation (p. 373)
3. Actor (p. 362)
4. Aggregation (p. 370)
5. Association (p. 366)
6. Association role (p. 366)
7. Behavior (p. 365)
8. Class diagram (p. 365)
9. Component diagram (p. 375)
10. Concrete class (p. 370)
11. Encapsulation (p. 366)
12. Event (p. 371)
13. Multiplicity (p. 366)
14. Object (p. 365)
15. Object class (p. 365)
16. Object diagram (p. 365)
17. Operation (p. 366)
18. Sequence diagram (p. 372)
19. Simple message (p. 374)
20. State (p. 365)
21. State transition (p. 371)
22. Synchronous message (p. 373)
23. Unified Modeling Language (UML) (p. 362)
24. Use case (p. 362)
25. Use-case diagram (p. 363)

Match each of the key terms above with the definition that best fits it.

____ 1. A diagram that depicts the use cases and actors for a system.
____ 2. Something that takes place at a certain point in time; it is a noteworthy occurrence that triggers a state transition.
____ 3. The time period during which an object performs an operation.
____ 4. A set of objects that share a common structure and a common behavior.
____ 5. A notation that allows the modeler to specify, visualize, and construct the artifacts of software systems, as well as business models.
____ 6. A type of message in which the caller has to wait for the receiving object to finish executing the called operation before it can resume execution itself.
____ 7. The end of an association where it connects to a class.
____ 8. A graph of instances that are compatible with a given class diagram.
____ 9. The changes in the attributes of an object or in the links an object has with other objects.
____ 10. An entity that has a well-defined role in the application domain and has state, behavior, and identity.
____ 11. A diagram that shows the software components or modules and their dependencies.
____ 12. An external entity that interacts with the system (similar to an external entity in data-flow diagramming).
____ 13. A complete sequence of related actions initiated by an actor; it represents a specific way to use the system.
____ 14. A *part-of* relationship between a component object and an aggregate object.
____ 15. An indication of how many objects participate in a given relationship.
____ 16. The technique of hiding the internal implementation details of an object from its external view.
____ 17. A function or a service that is provided by all the instances of a class.
____ 18. A condition that encompasses an object's properties (attributes and relationships) and the values those properties have.
____ 19. Represents how an object acts and reacts.
____ 20. A diagram that shows the static structure of an object-oriented model: the object classes, their internal structure, and the relationships in which they participate.
____ 21. A relationship among object classes.
____ 22. A class that has no direct instances but whose descendants may have direct instances.
____ 23. A class that can have direct instances.
____ 24. A depiction of the interactions among objects during a certain period of time.
____ 25. A message that transfers control from the sender to the recipient without describing the details of the communication.

Review Questions

1. Compare and contrast the object-oriented analysis and design models with structured analysis and design models.
2. Give an example of an abstract use case. Your example should involve at least two other use cases and show how they are related to the abstract use case.
3. Explain the use of association role for an association on a class diagram.
4. 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.
5. 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 the aggregation relationships.
6. Give an example of state transition. Your example should show how the state of the object undergoes a transition based on some event.

Problems and Exercises

1. The use-case diagram shown in Figure A-1 captures the Student billing function but does not contain any function for accepting tuition payment from students. Revise the diagram to capture this functionality. Also, express some common behavior among two use cases in the revised diagram by using include relationships.
2. Suppose that the employees of the university are not billed for tuition. Their spouses do not get a full-tuition waiver but pay for only 25 percent of