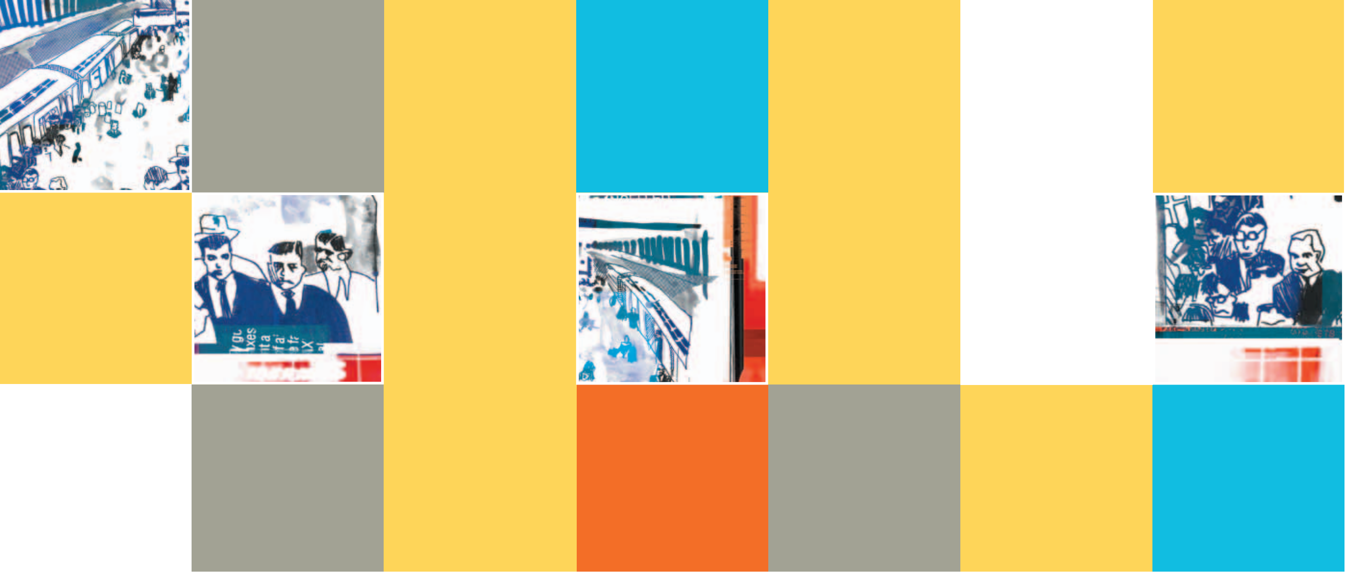


15 SYSTEM MODELING WITH THE UML

After studying Chapter 15, you should be able to:

- ☐ Understand the need for system modeling
- ☐ Describe the UML
- ☐ Work with use case diagrams
- ☐ Use class and object diagrams
- ☐ Use sequence and communication diagrams
- ☐ Use state machine diagrams
- ☐ Use activity diagrams
- ☐ Use component and deployment diagrams
- ☐ Diagram exception handling
- ☐ Decide which UML diagrams to use

UNDERSTANDING THE NEED FOR SYSTEM MODELING

Computer programs often stand alone to solve a user's specific problem. For example, a program might exist only to print paychecks for the current week. Most computer programs, however, are part of a larger system. Your company's payroll system might consist of dozens of programs, including programs that produce employee paychecks, apply raises to employee records, alter employee deduction options, and print W2 forms at the end of the tax year. Each program you write as part of a system might be related to several others. Some programs depend on input from other programs in the system or produce output to be fed into other programs. Similarly, an organization's accounting, inventory, and customer ordering systems all consist of many interrelated programs. Producing a set of programs that operate together correctly requires careful planning. **System design** is the detailed specification of how all the parts of a system will be implemented and coordinated.

TIP □ □ □ □ Usually, system design refers to computer system design, but even a noncomputerized, manual system can benefit from good design techniques.

Many textbooks cover the theories and techniques of system design. If you continue to study in a Computer Information Systems program at a college or university, you probably will be required to take a semester-long course in system design. Explaining all the techniques of system design is beyond the scope of this book. However, some basic principles parallel those you have used throughout this book in designing individual programs:

- Large systems are easier to understand when you break them down into subsystems.
- Good modeling techniques are increasingly important as the size and complexity of systems increase.
- Good models promote communication among technical and nontechnical workers while ensuring good business solutions.

In other words, developing a model for a single program or an entire business system requires organization and planning. In this chapter, you learn the basics of one popular design tool, the UML, which is based on these principles. The UML, or Unified Modeling Language, allows you to envision systems with an object-oriented perspective, breaking a system into subsystems, focusing on the big picture, and hiding the implementation details. In addition, the UML provides a means for programmers and businesspeople to communicate about system design. It also provides a way to plan to divide responsibilities for large systems. Understanding the UML's principles helps you design a variety of system types and talk about systems with the people who will use them.

TIP □ □ □ □ In addition to modeling a system before creating it, system analysts sometimes model an existing system to get a better picture of its operation. Creating a model for an existing system is called **reverse engineering**.

WHAT IS UML?

The **UML** is a standard way to specify, construct, and document systems that use object-oriented methods. (The UML is a modeling language, not a programming language. The systems you develop using the UML probably will be implemented later in object-oriented programming languages such as Java, C++, C#, or Visual Basic.) As with flowcharts, pseudocode, hierarchy charts, and class diagrams, the UML has its own notation that consists of a set of specialized shapes and conventions. You can use the UML's shapes to construct different kinds of software diagrams and model different kinds of systems. Just as you can use a flowchart or hierarchy chart to diagram real-life activities, organizational relationships, or computer programs, you also can use the UML for many purposes, including modeling business activities, organizational processes, or software systems.

TIP The UML was created at Rational Software by Grady Booch, Ivar Jacobson, and Jim Rumbaugh. The Object Management Group (OMG) adopted the UML as a standard for software modeling in 1997. The OMG includes more than 800 software vendors, developers, and users who seek a common architectural framework for object-oriented programming. The UML is in its second version, known as UML 2.0. You can view or download the entire UML specification and usage guidelines from the OMG at *www.uml.org*.

TIP You can purchase compilers for most programming languages from a variety of manufacturers. Similarly, you can purchase a variety of tools to help you create UML diagrams, but the UML itself is vendor-independent.

When you draw a flowchart or write pseudocode, your purpose is to illustrate the individual steps in a process. When you draw a hierarchy chart, you use more of a "big picture" approach. As with a hierarchy chart, you use the UML to create top-view diagrams of business processes that let you hide details and focus on functionality. This approach lets you start with a generic view of an application and introduce details and complexity later. UML diagrams are useful as you begin designing business systems, when customers who are not technically oriented must accurately communicate with the technical staff members who will create the actual systems. The UML was intentionally designed to be nontechnical so that developers, customers, and implementers (programmers) could all "speak the same language." If business and technical people can agree on what a system should do, the chances improve that the final product will be useful.

The UML is very large; its documentation is more than 800 pages. The UML provides 13 diagram types that you can use to model systems. Each of the diagram types lets you see a business process from a different angle, and appeals to a different type of user. Just as an architect, interior designer, electrician, and plumber use different diagram types to describe the same building, different computer users appreciate different perspectives. For example, a business user most values a system's use case diagrams because they illustrate who is doing what. On the other hand, programmers find class and object diagrams more useful because they help explain details of how to build classes and objects into applications.

The UML superstructure defines six structure diagrams, three behavior diagrams, and four interaction diagrams. The 13 UML diagram types are:

- **Structure diagrams**
  - Class diagrams
  - Object diagrams
  - Component diagrams
  - Composite structure diagrams
  - Package diagrams
  - Deployment diagrams
- **Behavior diagrams**
  - Use case diagrams
  - Activity diagrams
  - State machine diagrams
- **Interaction diagrams**
  - Sequence diagrams
  - Communication diagrams
  - Timing diagrams
  - Interaction overview diagrams

TIP You can categorize UML diagrams as those that illustrate the dynamic, or changing, aspects of a system and those that illustrate the static, or steady, aspects of a system. Dynamic diagrams include use case, sequence, communication, state machine, and activity diagrams. Static diagrams include class, object, component, and deployment diagrams.

TIP In UML 1.5, communication diagrams were called collaboration diagrams, and state machine diagrams were called statechart diagrams.

Each of the UML diagram types supports multiple variations, and explaining them all would require an entire textbook. This chapter presents an overview and simple examples of several diagram types, which provides a good foundation for further study of the UML.

TIP The UML Web site, at *www.uml.org*, provides links to several UML tutorials.

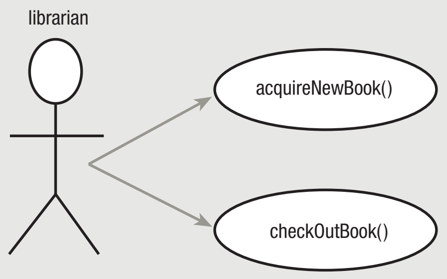
USING USE CASE DIAGRAMS

The **use case diagram** shows how a business works from the perspective of those who approach it from the outside, or those who actually use the business. This category includes many types of users—for example, employees, customers, and suppliers. Although users can also be governments, private organizations, machines, or other systems, it is easiest to think of them as people, so users are called actors and are represented by stick figures in use case diagrams. The actual use cases are represented by ovals.

Use cases do not necessarily represent all the functions of a system; they are the system functions or services that are visible to the system's actors. In other words, they represent the cases by which an actor uses and presumably benefits from the system. Determining all the cases for which users interact with systems helps you divide a system logically into functional parts.

Establishing use cases usually follows from analyzing the main events in a system. For example, from a librarian's point of view, two main events are `acquireNewBook()` and `checkOutBook()`. Figure 15-1 shows a use case diagram for these two events.

FIGURE 15-1: USE CASE DIAGRAM FOR LIBRARIAN



TIP

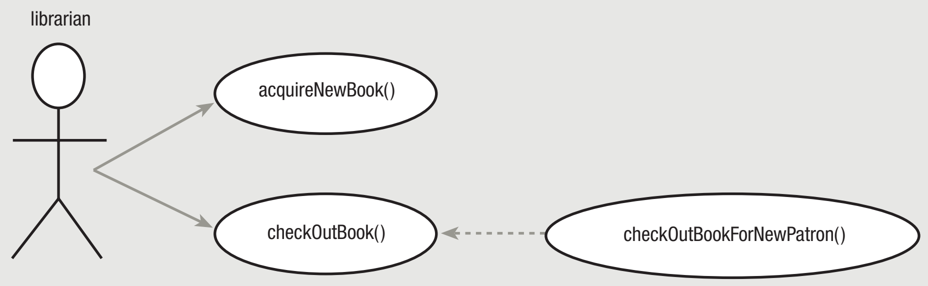
Many system developers would use the standard English form to describe activities in their UML diagrams—for example, `check out book` instead of `checkOutBook()`, which looks like a programming method call. Because you are used to seeing method names in camel casing and with trailing parentheses throughout this book, this discussion of the UML continues with the same format.

In many systems, there are variations in use cases. The three possible types of variations are:

- Extend
- Include
- Generalization

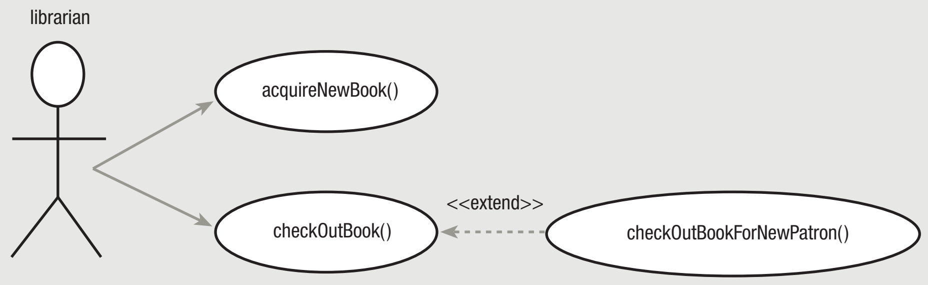
An **extend variation** is a use case variation that shows functions beyond those found in a base case. In other words, an extend variation is usually an optional activity. For example, checking out a book for a new library patron who doesn't have a library card is slightly more complicated than checking out a book for an existing patron. Each variation in the sequence of actions required in a use case is a **scenario**. Each use case has at least one main scenario, but might have several more that are extensions or variations of the main one. Figure 15-2 shows how you would diagram the relationship between the use case `checkOutBook()` and the more specific scenario `checkOutBookForNewPatron()`. Extended use cases are shown in an oval with a dashed arrow pointing to the more general base case.

FIGURE 15-2: USE CASE DIAGRAM FOR LIBRARIAN, WITH SCENARIO EXTENSION



For clarity, you can add "<<extend>>" near the line that shows a relationship extension. Such a feature, which adds to the UML vocabulary of shapes to make them more meaningful for the reader, is called a **stereotype**. Figure 15-3 includes a stereotype.

FIGURE 15-3: USE CASE DIAGRAM FOR LIBRARIAN, USING STEREOTYPE



In addition to extend relationships, use case diagrams also can show include relationships. You use an **include variation** when a case can be part of multiple use cases. This concept is very much like that of a subroutine or submodule. You show an include use case in an oval with a dashed arrow pointing to the subroutine use case. For example, `issueLibraryCard()` might be a function of `checkOutBook()`, which is used when the patron checking out a book is new, but it might also be a function of `registerNewPatron()`, which occurs when a patron registers at the library but does not want to check out books yet. See Figure 15-4.

You use a **generalization variation** when a use case is less specific than others, and you want to be able to substitute the more specific case for a general one. For example, a library has certain procedures for acquiring new materials, whether they are videos, tapes, CDs, hardcover books, or paperbacks. However, the procedures might become more specific during a particular acquisition—perhaps the librarian must procure plastic cases for circulating videos or assign locked storage locations for CDs. Figure 15-5 shows the generalization `acquireNewItem()` with two more specific situations: acquiring videos and acquiring CDs. The more specific scenarios are attached to the general scenario with open-headed dashed arrows.

FIGURE 15-4: USE CASE DIAGRAM FOR LIBRARIAN, USING INCLUDE RELATIONSHIP

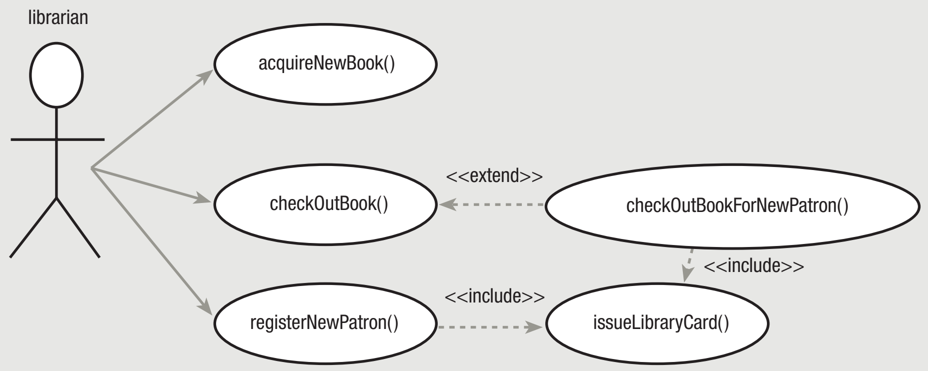
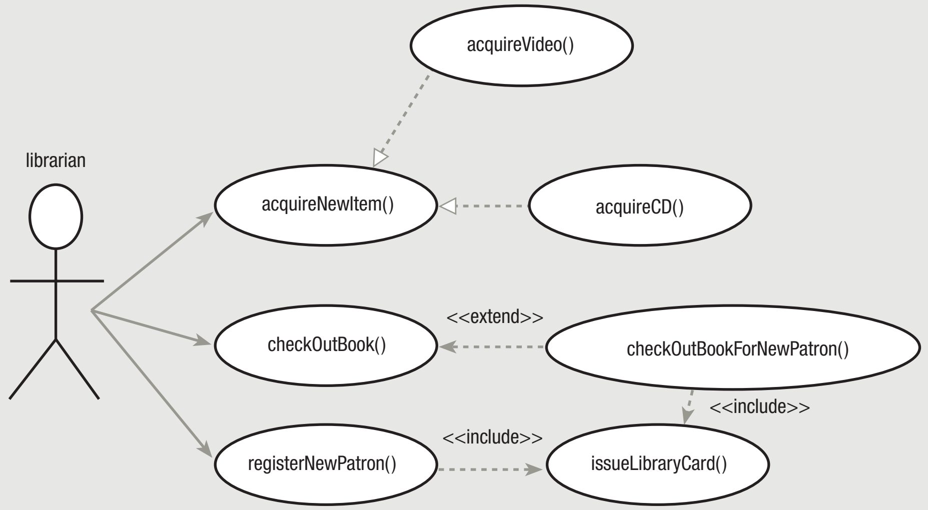
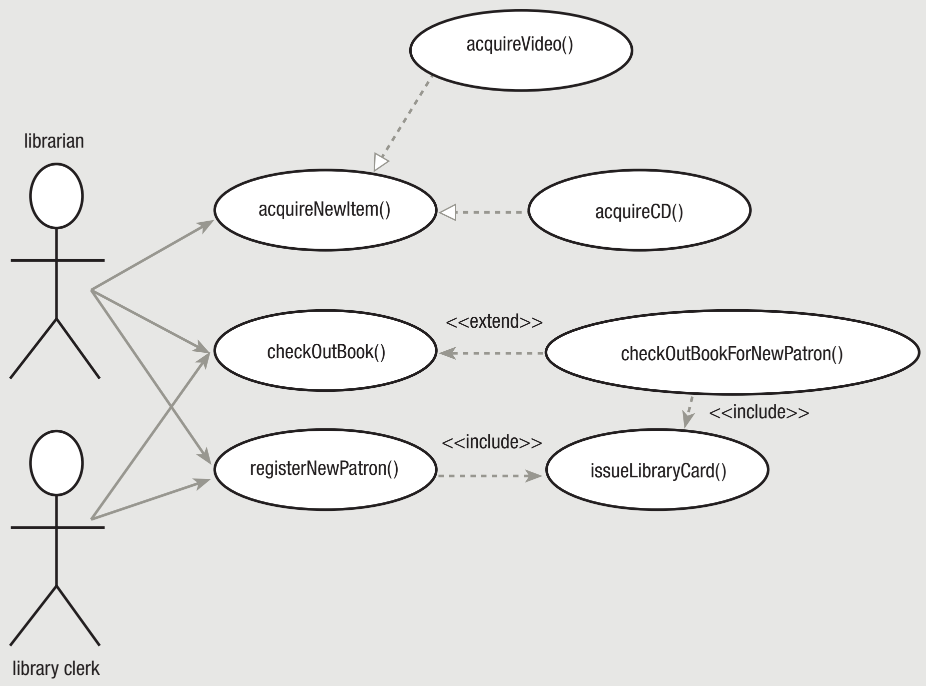


FIGURE 15-5: USE CASE DIAGRAM FOR LIBRARIAN, WITH GENERALIZATIONS



Many use case diagrams show multiple actors. For example, Figure 15-6 shows that a library clerk cannot perform as many functions as a librarian; the clerk can check out books and register new patrons but cannot acquire new materials.

FIGURE 15-6: USE CASE DIAGRAM FOR LIBRARIAN, WITH MULTIPLE ACTORS



While designing an actual library system, you could add many more use cases and actors to the use case diagram. The purpose of such a diagram is to encourage discussion between the system developer and the library staff. Library staff members do not need to know any of the technical details of the system that the analysts will eventually create, and they certainly do not need to understand computers or programming. However, by viewing the use cases, the library staff can visualize activities they perform while doing their jobs and correct the system developer if inaccuracies exist. The final software products developed for such a system are far more likely to satisfy users than those developed without this design step.

A use case diagram is only a tool to aid communication. No single "correct" use case diagram exists; you might correctly represent a system in several ways. For example, you might choose to emphasize the actors in the library system, as shown in Figure 15-7, or to emphasize system requirements, as shown in Figure 15-8. Diagrams that are too crowded are neither visually pleasing nor very useful. Therefore, the use case diagram in Figure 15-7 shows all the specific actors and their relationships, but purposely omits more specific system functions, whereas Figure 15-8 shows many actions that are often hidden from users but purposely omits more specific actors. For example, the activities carried out to `manageNetworkOutage()`, if done properly, should be invisible to library patrons checking out books.

FIGURE 15-7: USE CASE DIAGRAM EMPHASIZING ACTORS

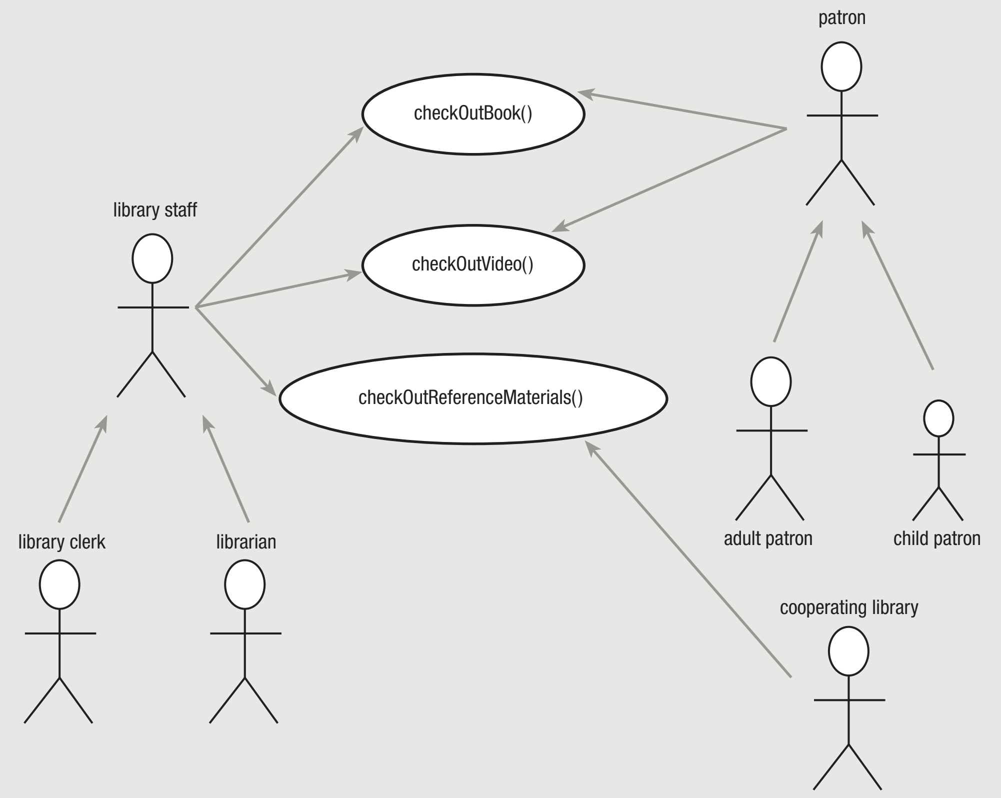
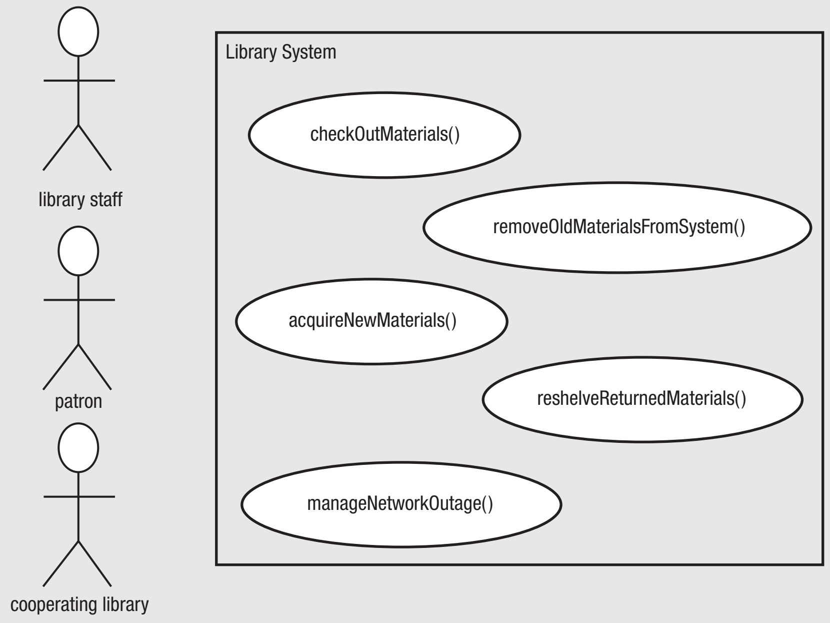


FIGURE 15-8: USE CASE DIAGRAM EMPHASIZING SYSTEM REQUIREMENTS



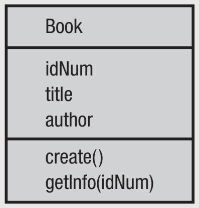
In Figure 15-8, the relationship lines between the actors and use cases have been removed because the emphasis is on the system requirements, and too many lines would make the diagram confusing. When system developers omit parts of diagrams for clarity, they refer to the missing parts as **elided**. For the sake of clarity, eliding extraneous information is perfectly acceptable. The main purpose of UML diagrams is to facilitate clear communication.

USING CLASS AND OBJECT DIAGRAMS

You use a class diagram to illustrate the names, attributes, and methods of a class or set of classes. Class diagrams are more useful to a system's programmers than to its users because they closely resemble code the programmers will write. A class diagram illustrating a single class contains a rectangle divided into three sections: the top section contains the name of the class, the middle section contains the names of the attributes, and the bottom section contains the names of the methods. Figure 15-9 shows the class diagram for a `Book` class. Each `Book` object contains an `idNum`, `title`, and `author`. Each `Book` object also contains methods to create a `Book` when it is acquired and to retrieve or get `title` and `author` information when the `Book` object's `idNum` is supplied.

TIP You first used class diagrams in Chapter 13.

FIGURE 15-9: `Book` CLASS DIAGRAM

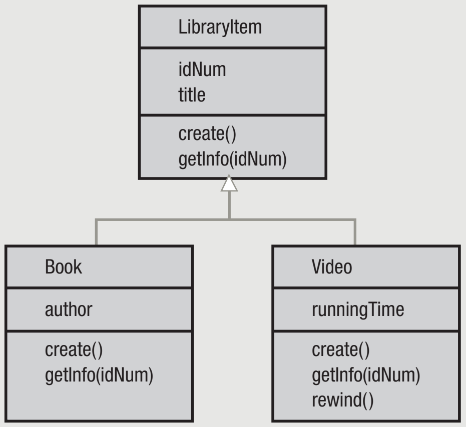


In the preceding section, you learned how to use generalizations with use case diagrams to show general and more specific use cases. With use case diagrams, you drew an open-headed arrow from the more specific case to the more general one. Similarly, you can use generalizations with class diagrams to show more general (or parent) classes and more specific (or child) classes that inherit attributes from parents. For example, Figure 15-10 shows `Book` and `Video` classes that are more specific than the general `LibraryItem` class. All `LibraryItem` objects contain an `idNum` and `title`, but each `Book` item also contains an `author`, and each `Video` item also contains a `runningTime`. In addition, `Video` items contain a `rewind()` method not found in the more general `LibraryItem` class. Child classes contain all the attributes of their parents and usually contain additional attributes not found in the parent.

TIP

You first learned about inheritance and parent and child classes in Chapter 13. There, you learned that the `create()` and `getInfo()` methods in the `Book` and `Video` classes override the version in the `LibraryItem` class.

FIGURE 15-10: `LibraryItem` CLASS DIAGRAM SHOWING GENERALIZATION



Class diagrams can include symbols that show the relationships between objects. You can show two types of relationships:

- An association relationship
- A whole-part relationship

An **association relationship** describes the connection or link between objects. You represent an association relationship between classes with a straight line. Frequently, you include information about the arithmetical relationship or ratio (called **cardinality** or **multiplicity**) of the objects. For example, Figure 15-11 shows the association relationship between a `Library` and the `LibraryItems` it lends. Exactly one `Library` object exists, and it can be associated with any number of `LibraryItems` from 0 to infinity, which is represented by an asterisk. Figure 15-12 adds the `Patron` class to the diagram and shows how you indicate that any number of `Patrons` can be associated with the `Library`, but that each `Patron` can borrow only up to five `LibraryItems` at a time, or currently might not be borrowing any. In addition, each `LibraryItem` can be associated with at most one `Patron`, but at any given time might not be on loan.

FIGURE 15-11: CLASS DIAGRAM WITH ASSOCIATION RELATIONSHIP

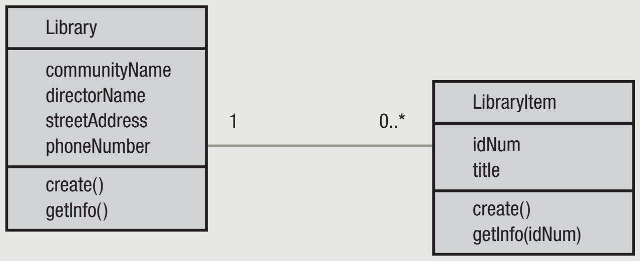
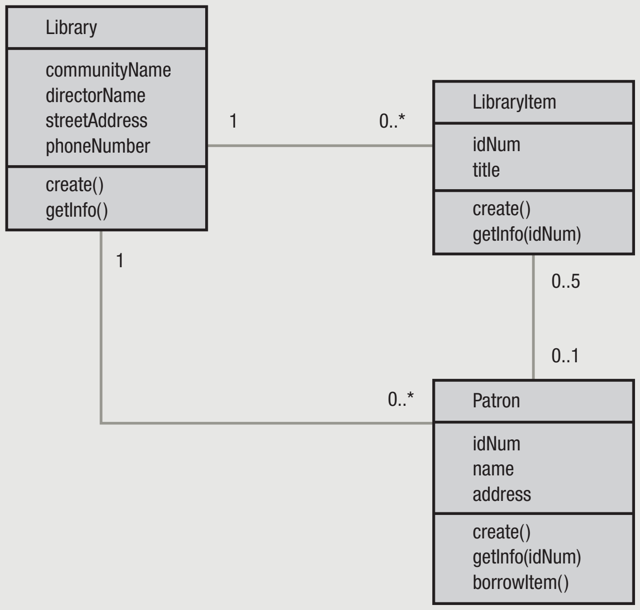
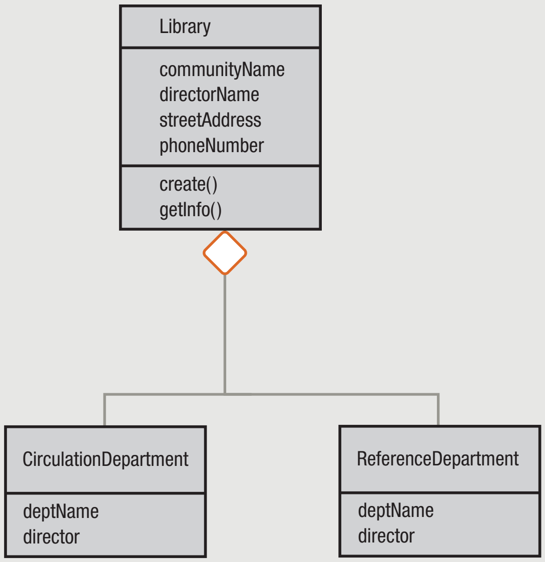


FIGURE 15-12: CLASS DIAGRAM WITH SEVERAL ASSOCIATION RELATIONSHIPS



A **whole-part relationship** describes an association in which one or more classes make up the parts of a larger whole class. For example, 50 states "make up" the United States, and 10 departments might "make up" a company. This type of relationship is also called an **aggregation** and is represented by an open diamond at the "whole part" end of the line that indicates the relationship. You also can call a whole-part relationship a **has-a relationship** because the phrase describes the association between the whole and one of its parts; for example, "The library has a Circulation Department." Figure 15-13 shows a whole-part relationship for a `Library`.

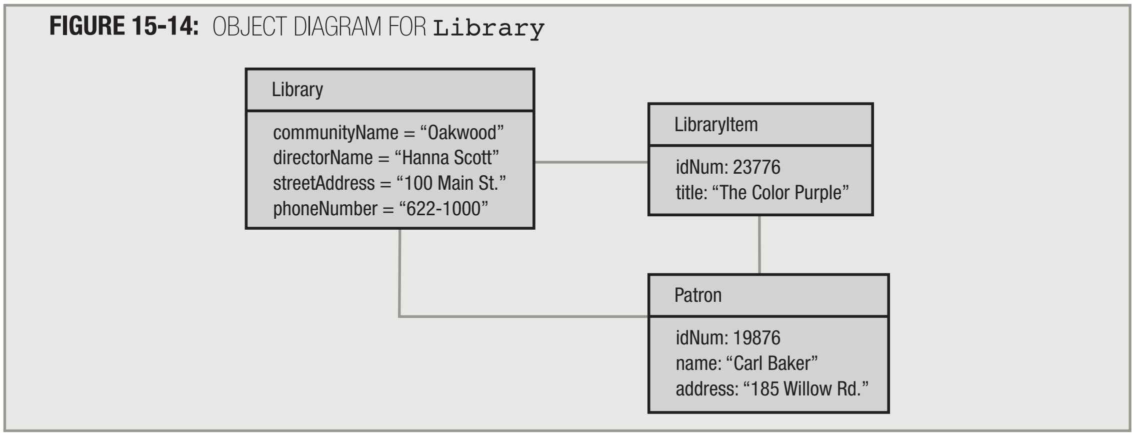
FIGURE 15-13: CLASS DIAGRAM WITH WHOLE-PART RELATIONSHIP



Object diagrams are similar to class diagrams, but they model specific instances of classes. You use an object diagram to show a snapshot of an object at one point in time, so you can more easily understand its relationship to other objects. Imagine looking at the travelers in a major airport. If you try to watch them all at once, you see a flurry of activity, but it is hard to understand all the tasks (buying a ticket, checking luggage, and so on) a traveler must accomplish to take a trip. However, if you concentrate on one traveler and follow his or her actions through the airport from arrival to takeoff, you get a clearer picture of the required activities. An object diagram serves the same purpose; you concentrate on a specific instance of a class to better understand how a class works.

Figure 15-14 contains an object diagram showing the relationship between one `Library`, `LibraryItem`, and `Patron`. Notice the similarities between Figures 15-12 and 15-14. If you need to describe the relationship between three classes, you can use either model—a class diagram or an object diagram—interchangeably. You simply use the model that seems clearer to you and your intended audience.

FIGURE 15-14: OBJECT DIAGRAM FOR `Library`

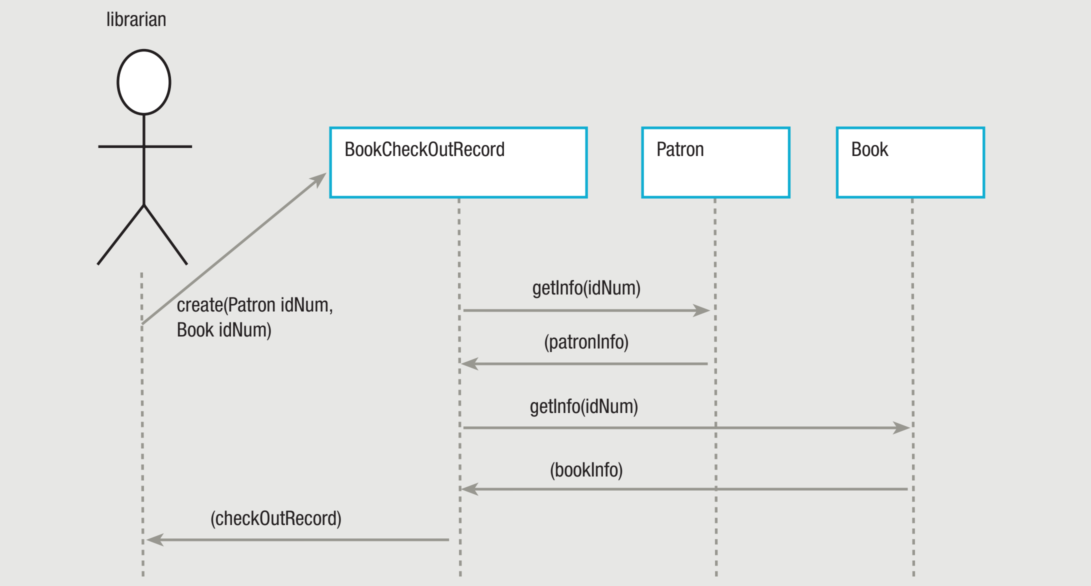

Library
communityName = “Oakwood”
directorName = “Hanna Scott”
streetAddress = “100 Main St.”
phoneNumber = “622-1000”
LibraryItem
idNum: 23776
title: “The Color Purple”
Patron
idNum: 19876
name: “Carl Baker”
address: “185 Willow Rd.”

USING SEQUENCE AND COMMUNICATION DIAGRAMS

You use a **sequence diagram** to show the timing of events in a single use case. A sequence diagram makes it easier to see the order in which activities occur. The horizontal axis (x-axis) of a sequence diagram represents objects, and the vertical axis (y-axis) represents time. You create a sequence diagram by placing objects that are part of an activity across the top of the diagram along the x-axis, starting at the left with the object or actor that begins the action. Beneath each object on the x-axis, you place a vertical dashed line that represents the period of time the object exists. Then, you use horizontal arrows to show how the objects communicate with each other over time.

For example, Figure 15-15 shows a sequence diagram for a scenario that a librarian can use to create a book check-out record. The librarian begins a `create()` method with `Patron idNum` and `Book idNum` information. The `BookCheckOutRecord` object requests additional `Patron` information (such as `name` and `address`) from the `Patron` object with the correct `Patron idNum`, and additional `Book` information (such as `title` and `author`) from the `Book` object with the correct `Book idNum`. When `BookCheckOutRecord` contains all the data it needs, a completed record is returned to the librarian.

FIGURE 15-15: SEQUENCE DIAGRAM FOR CHECKING OUT A `Book` FOR A `Patron`

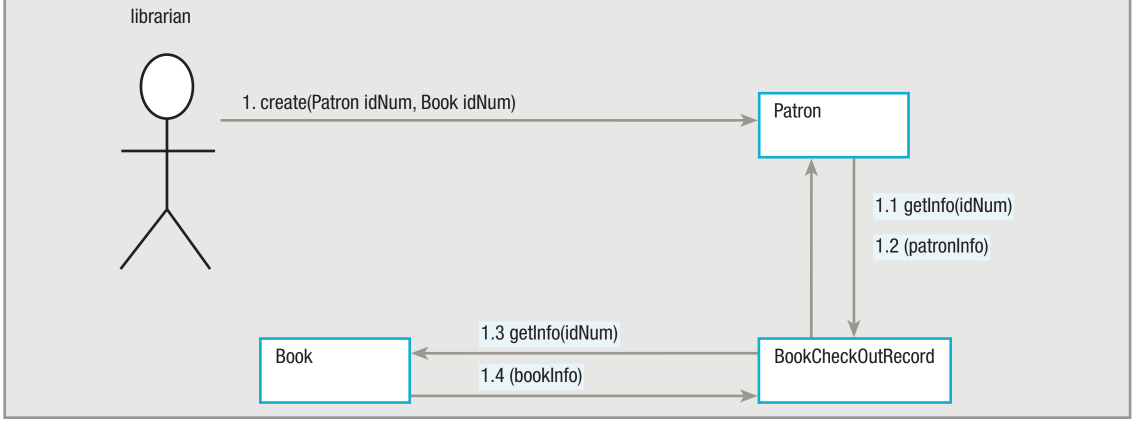


TIP

In Figures 15-15 and 15-16, `patronInfo` and `bookInfo` represent group items that contain all of a `Patron`'s and `Book`'s data. For example, `patronInfo` might contain `idNum`, `lastName`, `firstName`, `address`, and `phoneNumber`, all of which have been defined as attributes of that class.

A **communication diagram** emphasizes the organization of objects that participate in a system. It is similar to a sequence diagram, except that it contains sequence numbers to represent the precise order in which activities occur. Communication diagrams focus on object roles instead of the times that messages are sent. Figure 15-16 shows the same sequence of events as Figure 15-15, but the steps to creating a `BookCheckOutRecord` are clearly numbered (see the shaded sections of the figure). Decimal numbered steps (1.1, 1.2, and so on) represent substeps of the main steps. Checking out a library book is a fairly straightforward event, so a sequence diagram sufficiently illustrates the process. Communication diagrams become more useful with more complicated systems.

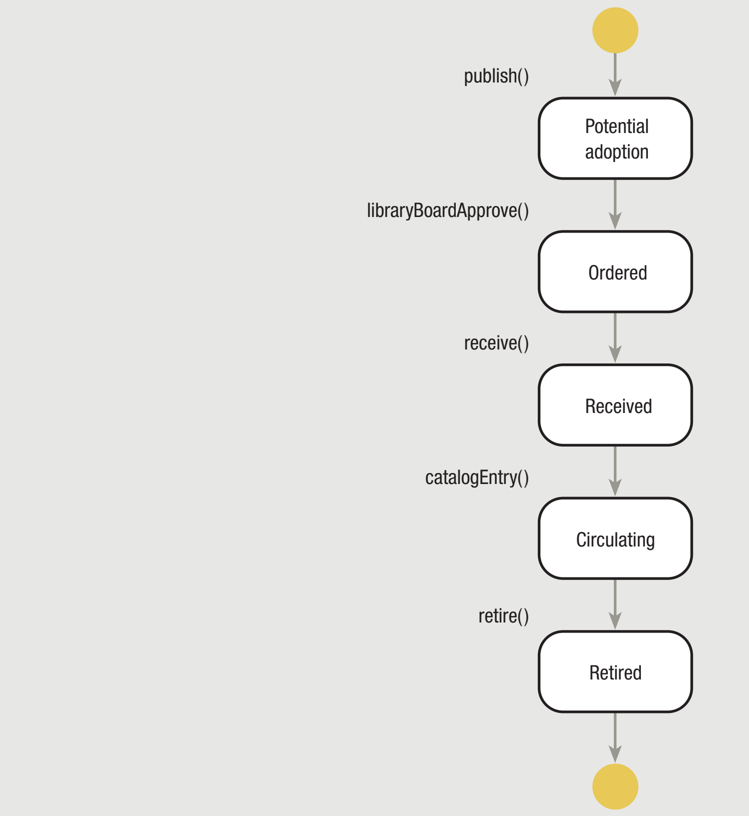
FIGURE 15-16: COMMUNICATION DIAGRAM FOR CHECKING OUT A `Book` FOR A `Patron`



USING STATE MACHINE DIAGRAMS

A **state machine diagram** shows the different statuses of a class or object at different points in time. You use a state machine diagram to illustrate aspects of a system that show interesting changes in behavior as time passes. Conventionally, you use rounded rectangles to represent each state and labeled arrows to show the sequence in which events affect the states. A solid dot indicates the start and stop states for the class or object. Figure 15-17 contains a state machine diagram you can use to describe the states of a `Book`.

FIGURE 15-17: STATE MACHINE DIAGRAM FOR `Book` CLASS



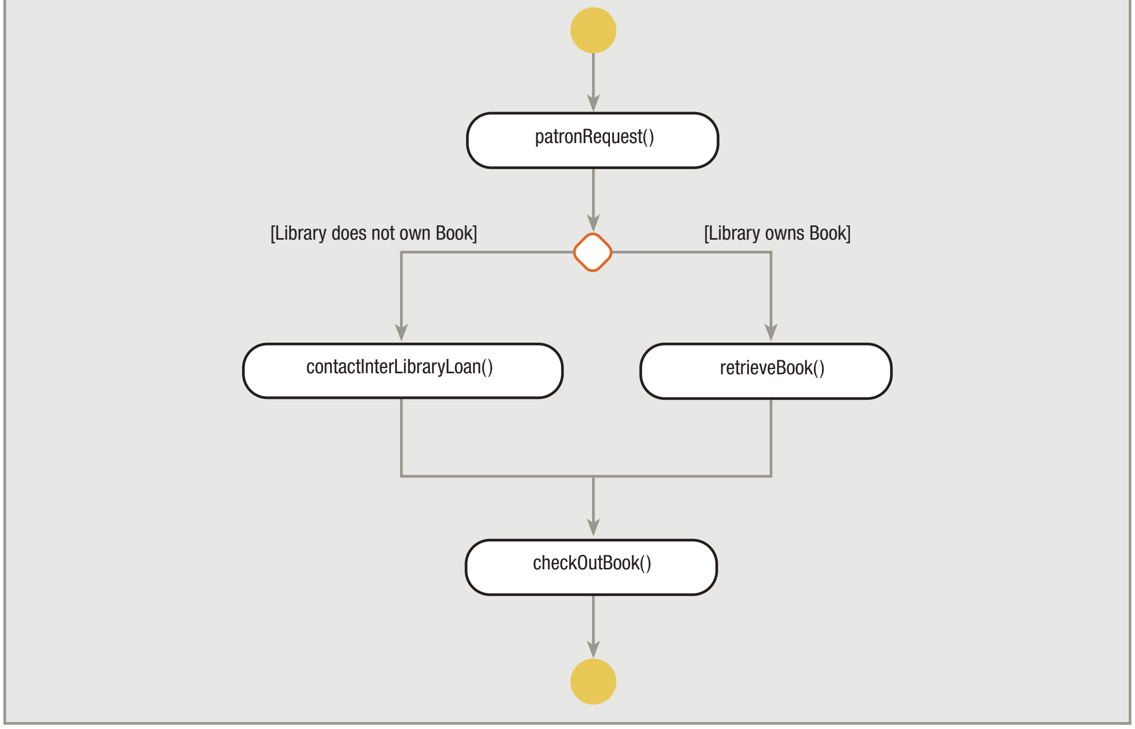
TIP

So that your diagrams are clear, you should use the correct symbol in each UML diagram you create, just as you should use the correct symbol in each program flowchart. However, if you create a flowchart and use a rectangle for an input or output statement where a parallelogram is conventional, others will still understand your meaning. Similarly, with UML diagrams, the exact shape you use is not nearly as important as the sequence of events and relationships between objects.

USING ACTIVITY DIAGRAMS

The UML diagram that most closely resembles a conventional flowchart is an activity diagram. In an **activity diagram**, you show the flow of actions of a system, including branches that occur when decisions affect the outcome. Conventionally, activity diagrams use flowchart start and stop symbols (called lozenges) to describe actions and solid dots to represent start and stop states. Like flowcharts, activity diagrams use diamonds to describe decisions. Unlike the diamonds in flowcharts, the diamonds in UML activity diagrams usually are empty; the possible outcomes are documented along the branches emerging from the decision symbol. As an example, Figure 15-18 shows a simple activity diagram with a single branch.

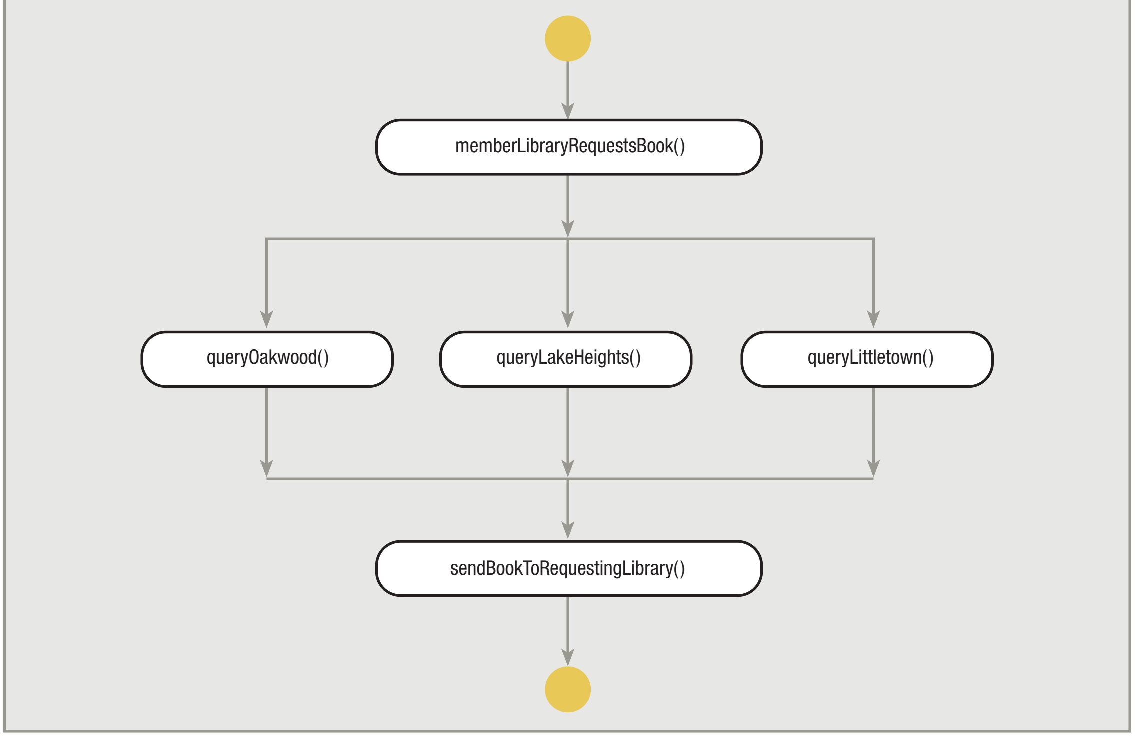
FIGURE 15-18: ACTIVITY DIAGRAM SHOWING BRANCH



TIP In the first version of the UML (UML 1.0), each lozenge was an activity. In the second version (UML 2.0), each lozenge is an action and a group of actions is an activity.

Many real-life systems contain actions that are meant to occur simultaneously. For example, when you apply for a home mortgage with a bank, a bank officer might perform a credit or background check while an appraiser determines the value of the house you are buying. When both actions are complete, the loan process continues. UML activity diagrams use forks and joins to show simultaneous activities. A fork is similar to a decision, but whereas the flow of control follows only one path after a decision, a fork defines a branch in which all paths are followed simultaneously. A join, as its name implies, reunites the flow of control after a fork. You indicate forks and joins with thick straight lines. Figure 15-19 shows how you might model the way an interlibrary loan system processes book requests. When a request is received, simultaneous searches begin at three local libraries that are part of the library system.

FIGURE 15-19: ACTIVITY DIAGRAM SHOWING FORK AND JOIN



TIP

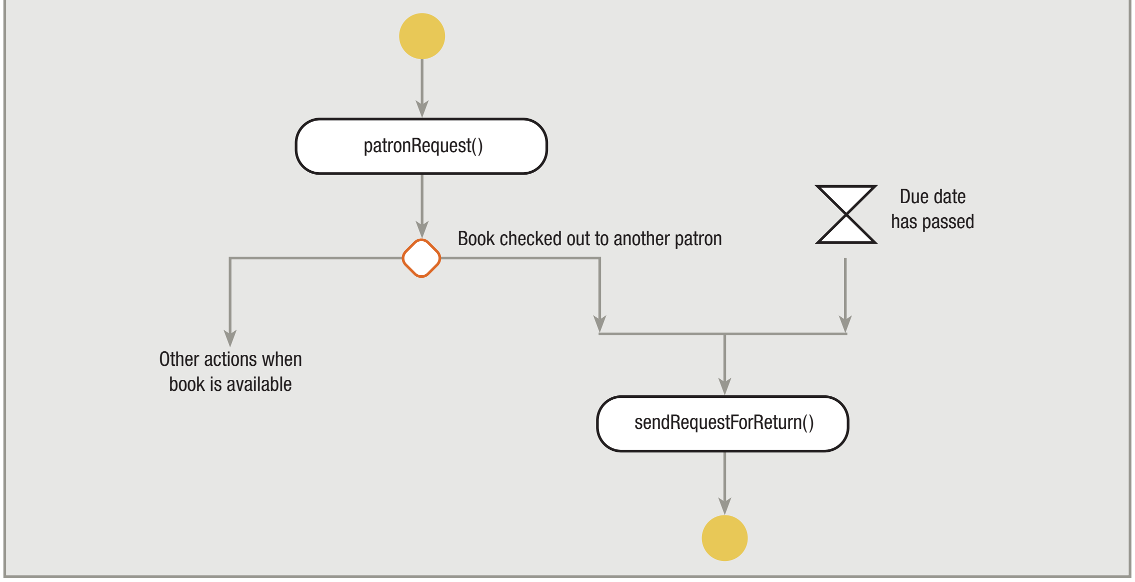
A fork does not have to indicate strictly simultaneous activity. The actions in the branches for a fork might only be concurrent or interleaved.

An activity diagram can contain a time signal. A **time signal** indicates that a specific amount of time has passed before an action is started. The time signal looks like two stacked triangles (resembling the shape of an hourglass). Figure 15-20 shows a time signal indicating that if a patron requests a book, and the book is checked out to another patron, then only if the book's due date has passed should a request to return the book be issued. In activity diagrams for other systems, you might see explanations at time signals, such as "10 hours have passed" or "at least January 1st". If an action is time-dependent, whether by a fraction of a second or by years, using a time signal is appropriate.

TIP

The time signal is a new feature in UML 2.0.

FIGURE 15-20: A TIME SIGNAL STARTING AN ACTION


patronRequest()
Book checked out to another patron
Due date has passed
Other actions when book is available
sendRequestForReturn()

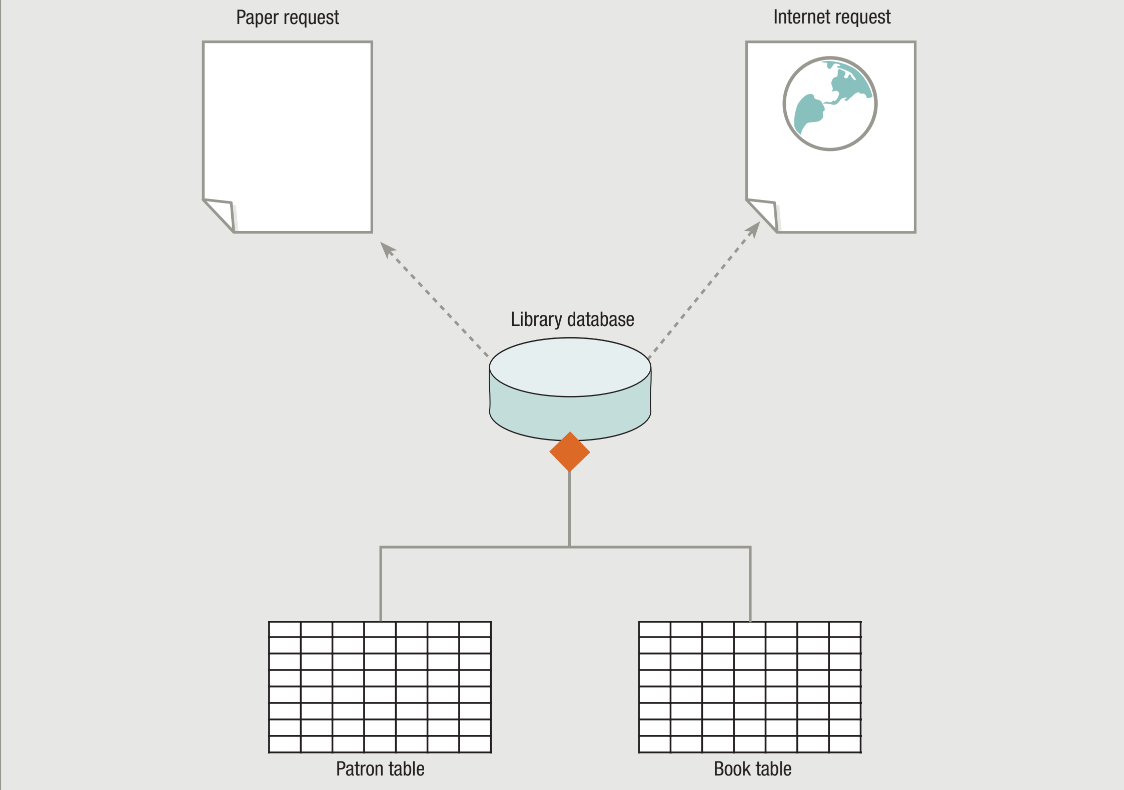
TIP

The connector is a recently introduced symbol to the UML. It is a small circle used to connect diagrams that are continued on a new page. It is identical to the flowchart connector symbol you learned about in Chapter 1.

USING COMPONENT AND DEPLOYMENT DIAGRAMS

Component and deployment diagrams model the physical aspects of systems. You use a **component diagram** when you want to emphasize the files, database tables, documents, and other components that a system's software uses. You use a **deployment diagram** when you want to focus on a system's hardware. You can use a variety of icons in each type of diagram, but each icon must convey meaning to the reader. Figures 15-21 and 15-22 show component and deployment diagrams that illustrate aspects of a library system. Figure 15-21 contains icons that symbolize paper and Internet requests for library items, the library database, and two tables that constitute the database. Figure 15-22 shows some commonly used icons that represent hardware components.

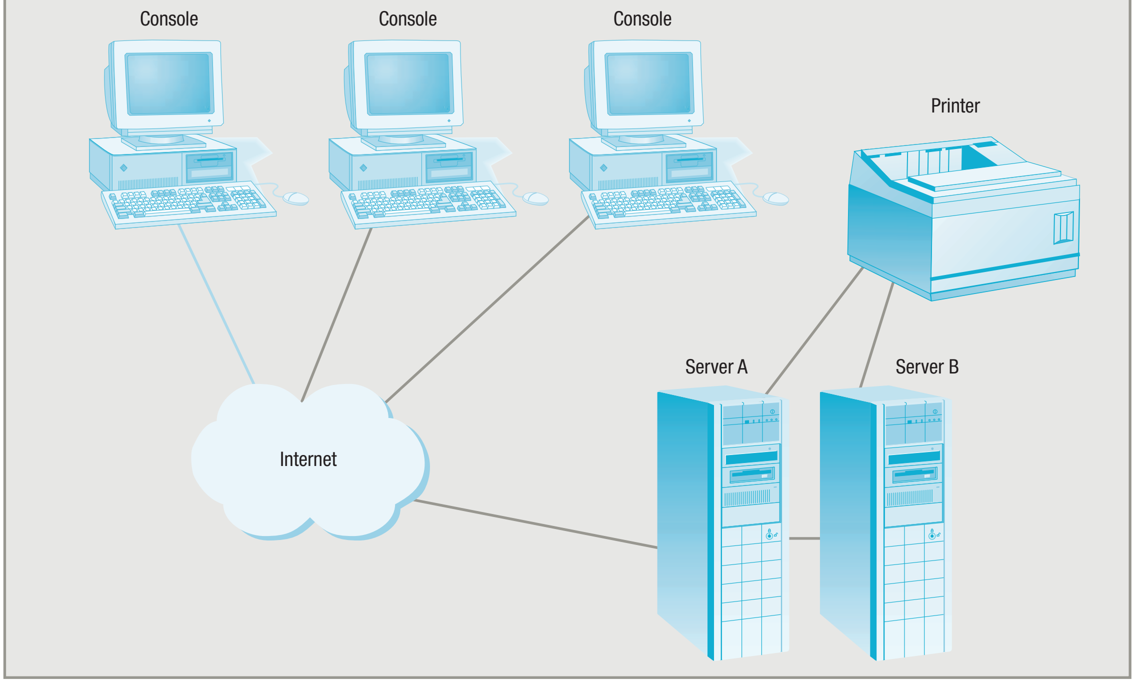
FIGURE 15-21: COMPONENT DIAGRAM



TIP

In Figure 15-21, notice the filled diamond connecting the two tables to the database. Just as it does in a class diagram, the diamond aggregation symbol shows the whole-part relationship of the tables to the database. You use an open diamond when a part might belong to several wholes (for example, `Door` and `Wall` objects belong to many `House` objects), but you use a filled diamond when a part can belong to only one whole at a time (the `Patron` table can belong only to the `Library` database). You can use most UML symbols in multiple types of diagrams.

FIGURE 15-22: DEPLOYMENT DIAGRAM



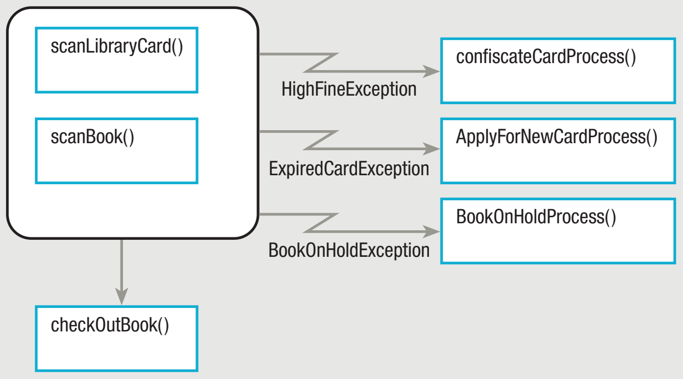
DIAGRAMMING EXCEPTION HANDLING

Exception handling is a set of the object-oriented techniques used to handle program errors. In Chapter 14, you learned that when a segment of code might cause an error, you can place that code in a `try` block. If the error occurs, an object called an exception is thrown, or sent, to a `catch` block where appropriate action can be taken. For example, depending on the application, a `catch` block might display a message, assign a default value to a field, or prompt the user for direction.

In the UML, a `try` block is called a **protected node** and a `catch` block is a **handler body node**. In a UML diagram, a protected node is enclosed in a rounded rectangle and any exceptions that might be thrown are listed next to lightning-bolt-shaped arrows that extend to the appropriate handler body node.

Figure 15-23 shows an example of an activity that uses exception handling. When a library patron tries to check out a book, the patron's card is scanned and the book is scanned. These actions might cause three errors—the patron owes fines, and so cannot check out new books; the patron's card has expired, requiring a new card application; or the book might be on hold for another patron. If no exceptions occur, the activity proceeds to the `checkOutBook()` process.

FIGURE 15-23: EXCEPTIONS IN THE BOOK CHECK-OUT ACTIVITY



DECIDING WHICH UML DIAGRAMS TO USE

Each of the UML diagram types provides a different view of a system. Just as a portrait artist, psychologist, and neurosurgeon each prefer a different conceptual view of your head, the users, managers, designers, and technicians of computer and business systems each prefer specific system views. Very few systems require diagrams of all 13 types; you can illustrate the objects and activities of many systems by using a single diagram, or perhaps one that is a hybrid of two or more basic types. No view is superior to the others; you can achieve the most complete picture of any system by using several views. The most important reason you use any UML diagram is to communicate clearly and efficiently with the people for whom you are designing a system.

CHAPTER SUMMARY

- System design is the detailed specification of how all the parts of a system will be implemented and coordinated. Good designs make systems easier to understand. The UML (Unified Modeling Language) provides a means for programmers and businesspeople to communicate about system design.
- The UML is a standard way to specify, construct, and document systems that use object-oriented methods. The UML has its own notation, with which you can construct software diagrams that model different kinds of systems. The UML provides 13 diagram types that you use at the beginning of the design process.
- A use case diagram shows how a business works from the perspective of those who approach it from the outside, or those who actually use the business. The diagram often includes actors, represented by stick figures, and use cases, represented by ovals. Use cases can include variations such as extend relationships, include relationships, and generalizations.
- You use a class diagram to illustrate the names, attributes, and methods of a class or set of classes. A class diagram of a single class contains a rectangle divided into three sections: the name of the class, the names of the attributes, and the names of the methods. Class diagrams can show generalizations and the relationships between objects. Object diagrams are similar to class diagrams, but they model specific instances of classes at one point in time.
- You use a sequence diagram to show the timing of events in a single use case. The horizontal axis (x-axis) of a sequence diagram represents objects, and the vertical axis (y-axis) represents time. A communication diagram emphasizes the organization of objects that participate in a system. It is similar to a sequence diagram, except that it contains sequence numbers to represent the precise order in which activities occur.
- A state machine diagram shows the different statuses of a class or object at different points in time.
- In an activity diagram, you show the flow of actions of a system, including branches that occur when decisions affect the outcome. UML activity diagrams use forks and joins to show simultaneous activities.
- You use a component diagram when you want to emphasize the files, database tables, documents, and other components that a system's software uses. You use a deployment diagram when you want to focus on a system's hardware.
- Each of the UML diagram types provides a different view of a system. Very few systems require diagrams of all 13 types; the most important reason to use any UML diagram is to communicate clearly and efficiently with the people for whom you are designing a system.

KEY TERMS

System design is the detailed specification of how all the parts of a system will be implemented and coordinated.

Reverse engineering is the process of creating a model of an existing system.

The **UML** is a standard way to specify, construct, and document systems that use object-oriented methods. UML is an acronym for Unified Modeling Language.

The **use case diagram** is a UML diagram that shows how a business works from the perspective of those who approach it from the outside, or those who actually use the business.

An **extend variation** is a use case variation that shows functions beyond those found in a base case.

Each variation in the sequence of actions required in a use case is a **scenario**.

A feature that adds to the UML vocabulary of shapes to make them more meaningful for the reader is called a **stereotype**.

An **include variation** is a use case variation that you use when a case can be part of multiple use cases in a UML diagram.

You use a **generalization variation** in a UML diagram when a use case is less specific than others, and you want to be able to substitute the more specific case for a general one.

When system developers omit parts of UML diagrams for clarity, they refer to the missing parts as **elided**.

An **association relationship** describes the connection or link between objects in a UML diagram.

Cardinality and **multiplicity** refer to the arithmetic relationships between objects.

A **whole-part relationship** describes an association in which one or more classes make up the parts of a larger whole class. This type of relationship is also called an **aggregation**. You also can call a whole-part relationship a **has-a relationship** because the phrase describes the association between the whole and one of its parts.

Object diagrams are UML diagrams that are similar to class diagrams, but they model specific instances of classes.

A **sequence diagram** is a UML diagram that shows the timing of events in a single use case.

A **communication diagram** is a UML diagram that emphasizes the organization of objects that participate in a system.

A **state machine diagram** is a UML diagram that shows the different statuses of a class or object at different points in time.

An **activity diagram** is a UML diagram that shows the flow of actions of a system, including branches that occur when decisions affect the outcome.

A **time signal** is a UML diagram symbol that indicates that a specific amount of time has passed before an action is started.

A **component diagram** is a UML diagram that emphasizes the files, database tables, documents, and other components that a system's software uses.

A **deployment diagram** is a UML diagram that focuses on a system's hardware.

A **protected node** is the UML diagram name for an exception-throwing `try` block.

A **handler body node** is the UML diagram name for an exception-handling `catch` block.

REVIEW QUESTIONS

1. **The detailed specification of how all the parts of a system will be implemented and coordinated is called __________.**
   a. programming
   b. paraphrasing
   c. system design
   d. structuring

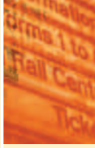
2. **The primary purpose of good modeling techniques is to __________.**
   a. promote communication
   b. increase functional cohesion
   c. reduce the need for structure
   d. reduce dependency between modules

3. **The Unified Modeling Language provides standard ways to do all of the following to business systems except to __________ them.**
   a. construct
   b. document
   c. describe
   d. destroy

4. **The UML is commonly used to model all of the following except __________.**
   a. computer programs
   b. business activities
   c. organizational processes
   d. software systems

5. **The UML was intentionally designed to be __________.**
   a. low-level, detail-oriented
   b. used with Visual Basic
   c. nontechnical
   d. inexpensive

6. **The UML diagrams that show how a business works from the perspective of those who actually use the business, such as employees or customers, are __________ diagrams.**
   a. communication
   b. use case
   c. state machine
   d. class



7. **Which of the following is an example of a relationship that would be portrayed as an extend relationship in a use case diagram for a hospital?**
   a. the relationship between the head nurse and the floor nurses
   b. admitting a patient who has never been admitted before
   c. serving a meal
   d. scheduling the monitoring of patients' vital signs

8. **The people shown in use case diagrams are called __________.**
   a. workers
   b. clowns
   c. actors
   d. relatives

9. **One aspect of use case diagrams that makes them difficult to learn about is that __________.**
   a. they require programming experience to understand
   b. they use a technical vocabulary
   c. there is no single right answer for any case
   d. all of the above

10. **The arithmetic association relationship between a college student and college courses would be expressed as __________.**
    a. 1 0
    b. 1 1
    c. 1 0..*
    d. 0..* 0..*

11. **In the UML, object diagrams are most similar to __________ diagrams.**
    a. use case
    b. activity
    c. class
    d. sequence

12. **In any given situation, you should choose the type of UML diagram that is __________.**
    a. shorter than others
    b. clearer than others
    c. more detailed than others
    d. closest to the programming language you will use to implement the system

13. **A whole-part relationship can be described as a(n) __________ relationship.**
    a. parent-child
    b. is-a
    c. has-a
    d. creates-a

14. The timing of events is best portrayed in a(n) __________ diagram.

a. sequence
b. use case
c. communication
d. association

15. A communication diagram is closest to a(n) __________ diagram.

a. activity
b. use case
c. deployment
d. sequence

16. A(n) __________ diagram shows the different statuses of a class or object at different points in time.

a. activity
b. state machine
c. sequence
d. deployment

17. The UML diagram that most closely resembles a conventional flowchart is a(n) __________ diagram.

a. activity
b. state machine
c. sequence
d. deployment

18. You use a __________ diagram when you want to emphasize the files, database tables, documents, and other components that a system's software uses.

a. state machine
b. component
c. deployment
d. use case

19. The UML diagram that focuses on a system's hardware is a(n) __________ diagram.

a. deployment
b. sequence
c. activity
d. use case

20. When using the UML to describe a single system, most designers would use __________.

a. a single type of diagram
b. at least three types of diagrams
c. most of the available types of diagrams
d. all 13 types of diagrams

FIND THE BUGS

Because of the nature of this chapter, there are no debugging exercises.

EXERCISES

1. **Complete the following tasks:**
   a. Develop a use case diagram for a convenience food store. Include an actor representing the store manager and use cases for `orderItem()`, `stockItem()`, and `sellItem()`.
   b. Add more use cases to the diagram you created in Exercise 1a. Include two generalizations for `stockItem()`: `stockPerishable()` and `stockNonPerishable()`. Also include an extension to `sellItem()` called `checkCredit()` for when a customer purchases items using a credit card.
   c. Add a customer actor to the use case diagram you created in Exercise 1b. Show that the customer participates in `sellItem()`, but not in `orderItem()` or `stockItem()`.
2. **Develop a use case diagram for a department store credit card system. Include at least two actors and four use cases.**
3. **Develop a use case diagram for a college registration system. Include at least three actors and five use cases.**
4. **Develop a class diagram for a `Video` class that describes objects a video store customer can rent. Include at least four attributes and three methods.**
5. **Develop a class diagram for a `Shape` class. Include generalizations for child classes `Rectangle`, `Circle`, and `Triangle`.**
6. **Develop a class diagram for a `BankLoan` class. Include generalizations for child classes `Mortgage`, `CarLoan`, and `EducationLoan`.**
7. **Develop a class diagram for a college registration system. Include at least three classes that cooperate to achieve student registration.**
8. **Develop a sequence diagram that shows how a clerk at a mail-order company places a customer `Order`. The `Order` accesses `Inventory` to check availability. Then, the `Order` accesses `Invoice` to produce a customer invoice that returns to the clerk.**
9. **Develop a state machine diagram that shows the states of a `CollegeStudent` from `PotentialApplicant` to `Graduate`.**
10. **Develop a state machine diagram that shows the states of a `Book` from `Concept` to `Publication`.**
11. **Develop an activity diagram that illustrates how to build a house.**
12. **Develop an activity diagram that illustrates how to prepare dinner.**

13. **Develop the UML diagram of your choice that illustrates some aspect of your life.**

14. **Complete the following tasks:**
    a. Develop the UML diagram of your choice that best illustrates some aspect of a place you have worked.
    b. Develop a different UML diagram type that illustrates the same functions as the diagram you created in Exercise 14a.

DETECTIVE WORK

1. **What are the education requirements for a career in system design? What are the job prospects and average salaries?**

2. **Find any discussion you can on the advantages and disadvantages of the UML as a system design tool. Summarize your findings.**

UP FOR DISCUSSION

1. **Which do you think you would enjoy doing more on the job—designing large systems that contain many programs, or writing the programs themselves? Why?**

2. **In Chapter 11, you considered ethical dilemmas in writing a program that selects candidates for organ transplants. Are the ethical responsibilities of a system designer different from those of a programmer? If so, how?**