CHAPTER 5

Structural Modeling

A structural, or conceptual, model describes the structure of the objects that support the business processes in an organization. During analysis, the structural model presents the logical organization of the objects without indicating how they are stored, created, or manipulated so that analysts can focus on the business, without being distracted by technical details. Later during design, the structural model is updated to reflect exactly how the objects will be stored in databases and files. This chapter describes *class–responsibility–collaboration (CRC)* cards, *class diagrams,* and *object diagrams,* which are used to create the structural model.

OBJECTIVES

- Understand the rules and style guidelines for creating CRC cards, class diagrams, and object diagrams.
- Understand the processes used to create CRC cards, class diagrams, and object diagrams.
- Be able to create CRC cards, class diagrams, and object diagrams.
- Understand the relationship among the structural models.
- Understand the relationship between the structural and functional models.

INTRODUCTION

During analysis, analysts create business process and functional models to represent how the business system will behave. At the same time, analysts need to understand the information that is used and created by the business system (e.g., customer information, order information). In this chapter, we discuss how the objects underlying the behavior modeled in the business process and functional models are organized and presented.

As pointed out in Chapter 1, all object-oriented systems development approaches are use-case driven, architecture-centric, and iterative and incremental. Use cases, described in Chapter 4, form the foundation on which the business information system is created. From an architecture-centric perspective, structural modeling supports the creation of an internal structural or static view of a business information system in that it shows how the system is structured to support the underlying business processes. Finally, as with business process and functional modeling, you will find that you will need to not only iterate across the structural models (described in this chapter), but you will also have to iterate across all three architectural views (functional, structural, and behavioral) to fully capture and represent the requirements for a business information system.

A *structural model* is a formal way of representing the objects that are used and created by a business system. It illustrates people, places, or things about which information is captured and how they are related to one another. The structural model is drawn using an iterative process in which the model becomes more detailed and less conceptual over time. In analysis, analysts draw a *conceptual model,* which shows the logical organization of the objects without indicating how the objects are stored, created, or manipulated. Because this model is free from any implementation or technical details, the analysts can focus more easily on matching the model to the real business requirements of the system.

In design, analysts evolve the conceptual structural model into a design model that reflects how the objects will be organized in databases and software. At this point, the model is checked for redundancy, and the analysts investigate ways to make the objects easy to retrieve. The specifics of the design model are discussed in detail in the design chapters.

STRUCTURAL MODELS

Every time a systems analyst encounters a new problem to solve, the analyst must learn the underlying problem domain. The goal of the analyst is to discover the key objects contained in the problem domain and to build a structural model. Object-oriented modeling allows the analyst to reduce the semantic gap between the underlying problem domain and the evolving structural model. However, the real world and the world of software are very different. The real world tends to be messy, whereas the world of software must be neat and logical. Thus, an exact mapping between the structural model and the problem domain may not be possible. In fact, it might not even be desirable.

One of the primary purposes of the structural model is to create a vocabulary that can be used by the analyst and the users. Structural models represent the things, ideas, or concepts contained in the domain of the problem. They also allow the representation of the relationships among the things, ideas, or concepts. By creating a structural model of the problem domain, the analyst creates the vocabulary necessary for the analyst and users to communicate effectively.

It is important to remember that at this stage of development, the structural model does not represent software components or classes in an object-oriented programming language, even though the structural model does contain analysis classes, attributes, operations, and the relationships among the analysis classes. The refinement of these initial classes into programming-level objects comes later. Nonetheless, the structural model at this point should represent the responsibilities of each class and the collaborations among the classes. Typically, structural models are depicted using CRC cards, class diagrams, and, in some cases, object diagrams. However, before describing CRC cards, class diagrams, and object diagrams, we describe the basic elements of structural models: classes, attributes, operations, and relationships.

Classes, Attributes, and Operations

A *class* is a general template that we use to create specific instances, or *objects,* in the problem domain. All objects of a given class are identical in structure and behavior but contain different data in their attributes. There are two general kinds of classes of interest during analysis: concrete and abstract. Normally, when an analyst describes the application domain classes, he or she is referring to concrete classes; that is, *concrete classes* are used to create objects. *Abstract classes* do not actually exist in the real world; they are simply useful abstractions. For example, from an employee class and a customer class, we may identify a generalization of the two classes and name the abstract class *person.* We might not actually instantiate the person class in the system itself, instead creating and using only employees and customers.[1]

[1] Because abstract classes are essentially not necessary and are not instantiated, arguments have been made that it would be better not to include any of them in the description of the evolving system at this stage of development (see J. Evermann and Y. Wand, "Towards Ontologically Based Semantics for UML Constructs," in H. S. Junii, S. Jajodia, and A. Solvberg (eds.) *ER 2001, Lecture Notes in Computer Science 2224* (Berlin: Springer-Verlag, 2001): 354–367). However, because abstract classes traditionally have been included at this stage of development, we also include them.

A second classification of classes is the type of real-world thing that a class represents. There are domain classes, user-interface classes, data structure classes, file structure classes, operating environment classes, document classes, and various types of multimedia classes. At this point in the development of our evolving system, we are interested only in domain classes. Later in design and implementation, the other types of classes become more relevant.

An *attribute* of an analysis class represents a piece of information that is relevant to the description of the class within the application domain of the problem being investigated. An attribute contains information the analyst or user feels the system should keep track of. For example, a possible relevant attribute of an employee class is employee name, whereas one that might not be as relevant is hair color. Both describe something about an employee, but hair color is probably not all that useful for most business applications. Only attributes that are important to the task should be included in the class. Finally, only attributes that are primitive or atomic types (i.e., integers, strings, doubles, date, time, Boolean, etc.) should be added. Most complex or compound attributes are really placeholders for relationships between classes. Therefore, they should be modeled as relationships, not as attributes (see the next section).

The behavior of an analysis class is defined in an *operation* or service. In later phases, the operations are converted to *methods*. However, because methods are more related to implementation, at this point in the development we use the term *operation* to describe the actions to which the *instances* of the class are capable of responding. Like attributes, only problem domain–specific operations that are relevant to the problem being investigated should be considered. For example, it is normally required that classes provide means of creating instances, deleting instances, accessing individual attribute values, setting individual attribute values, accessing individual relationship values, and removing individual relationship values. However, at this point in the development of the evolving system, the analyst should avoid cluttering up the definition of the class with these basic types of operations and focus only on relevant problem domain–specific operations.

Relationships

There are many different types of relationships that can be defined, but all can be classified into three basic categories of data abstraction mechanisms: generalization relationships, aggregation relationships, and association relationships. These data-abstraction mechanisms allow the analyst to focus on the important dimensions while ignoring nonessential dimensions. As with attributes, the analyst must be careful to include only relevant relationships.

Generalization Relationships The generalization abstraction enables the analyst to create classes that inherit attributes and operations of other classes. The analyst creates a *superclass* that contains basic attributes and operations that will be used in several *subclasses*. The subclasses inherit the attributes and operations of their superclass and can also contain attributes and operations that are unique just to them. For example, a customer class and an employee class can be generalized into a person class by extracting the attributes and operations both have in common and placing them into the new superclass, *person*. In this way, the analyst can reduce the redundancy in the class definitions so that the common elements are defined once and then reused in the subclasses. Generalization is represented with the *a-kind-of* relationship, so that we say that an employee is a-kind-of person.

The analyst also can use the opposite of generalization. Specialization uncovers additional classes by allowing new subclasses to be created from an existing class. For example, an employee class can be specialized into a secretary class and an engineer class.

Furthermore, generalization relationships between classes can be combined to form generalization hierarchies. Based on the previous examples, a secretary class and an engineer class can be subclasses of an employee class, which in turn could be a subclass of a person class. This would be read as a secretary and an engineer are a-kind-of employee and a customer and an employee are a-kind-of person.

The generalization data abstraction is a very powerful mechanism that encourages the analyst to focus on the properties that make each class unique by allowing the similarities to be factored into superclasses. However, to ensure that the semantics of the subclasses are maintained, the analyst should apply the principle of *substitutability.* By this we mean that the subclass should be capable of substituting for the superclass anywhere that uses the superclass (e.g., anywhere we use the employee superclass, we could also logically use its secretary subclass). By focusing on the a-kind-of interpretation of the generalization relationship, the principle of substitutability is applied.

Aggregation Relationships Generally speaking, all aggregation relationships relate *parts* to *wholes* or *assemblies.* For our purposes, we use the *a-part-of* or *has-parts* semantic relationship to represent the aggregation abstraction. For example, a door is a-part-of a car, an employee is a-part-of a department, or a department is a-part-of an organization. Like the generalization relationship, aggregation relationships can be combined into aggregation hierarchies. For example, a piston is a-part-of an engine, and an engine is a-part-of a car.

Aggregation relationships are bidirectional. The flip side of aggregation is *decomposition.* The analyst can use decomposition to uncover parts of a class that should be modeled separately. For example, if a door and an engine are a-part-of a car, then a car has-parts door and engine. The analyst can bounce around between the various parts to uncover new parts. For example, the analyst can ask, What other parts are there to a car? or To which other assemblies can a door belong?

Association Relationships There are other types of relationships that do not fit neatly into a generalization (a-kind-of) or aggregation (a-part-of) framework. Technically speaking, these relationships are usually a weaker form of the aggregation relationship. For example, a patient schedules an appointment. It could be argued that a patient is a-part-of an appointment. However, there is a clear semantic difference between this type of relationship and one that models the relationship between doors and cars or even workers and unions. Thus, they are simply considered to be *associations* between instances of classes.

OBJECT IDENTIFICATION

Different approaches have been suggested to aid the analyst in identifying a set of candidate objects for the structural model. The four most common approaches are textual analysis, brainstorming, common object lists, and patterns. Most analysts use a combination of these techniques to make sure that no important objects and object attributes, operations, and relationships have been overlooked.

Textual Analysis

The analyst performs *textual analysis* by reviewing the use-case diagrams and examining the text in the use-case descriptions to identify potential objects, attributes, operations, and relationships. The nouns in the use case suggest possible classes, and the verbs suggest possible operations. Figure 5-1 presents a summary of useful guidelines. The textual analysis of

- A common or improper noun implies a class of objects.
- A proper noun or direct reference implies an instance of a class.
- A collective noun implies a class of objects made up of groups of instances of another class.
- An adjective implies an attribute of an object.
- A doing verb implies an operation.
- A being verb implies a classification relationship between an object and its class.
- A having verb implies an aggregation or association relationship.
- A transitive verb implies an operation.
- An intransitive verb implies an exception.
- A predicate or descriptive verb phrase implies an operation.
- An adverb implies an attribute of a relationship or an operation.

Adapted from: These guidelines are based on Russell J. Abbott, "Program Design by Informal English Descriptions," *Communications of the ACM* 26, no. 11 (1983): 882–894; Peter P-S Chen, "English Sentence Structure and Entity-Relationship Diagrams," *Information Sciences: An International Journal* 29, no. 2–3 (1983): 127–149; Ian Graham, *Migrating to Object Technology* (Reading, MA: Addison Wesley Longman, 1995).

FIGURE 5-1
Textual Analysis Guidelines

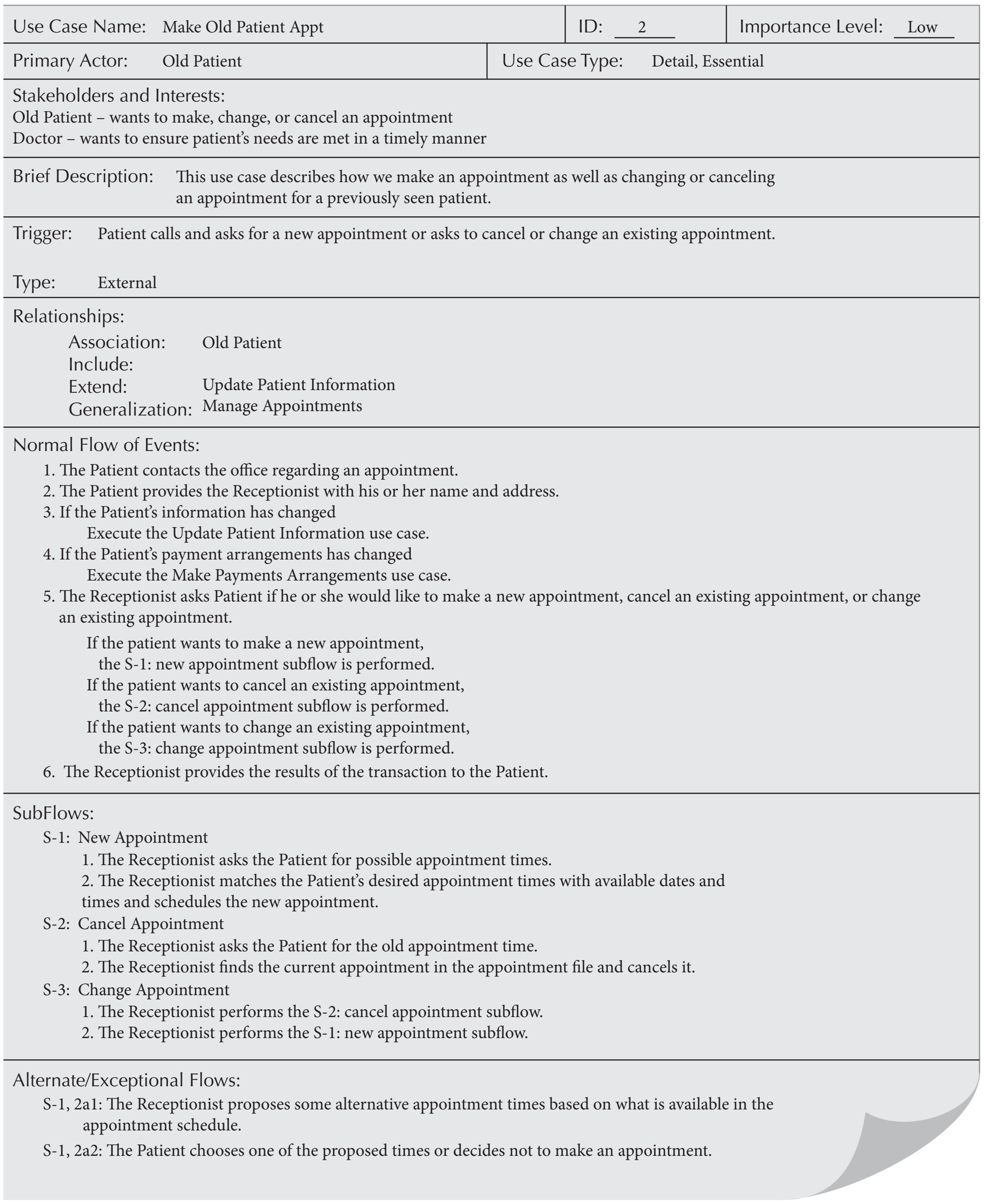
use-case descriptions has been criticized as being too simple, but because its primary purpose is to create an initial rough-cut structural model, its simplicity is a major advantage. For example, if we applied these rules to the Make Old Patient Appt use case described in Chapter 4 and replicated in Figure 5-2, we can easily identify potential objects for an old patient, doctor, appointment, patient, office, receptionist, name, address, patient information, payment, date, and time. We also can easily identify potential operations that can be associated with the identified objects. For example, patient contacts office, makes a new appointment, cancels an existing appointment, changes an existing appointment, matches requested appointment times and dates with requested times and dates, and finds current appointment.

Brainstorming

Brainstorming is a discovery technique that has been used successfully in identifying candidate classes. Essentially, in this context, brainstorming is a process that a set of individuals sitting around a table suggest potential classes that could be useful for the problem under consideration. Typically, a brainstorming session is kicked off by a facilitator who asks the set of individuals to address a specific question or statement that frames the session. For example, using the appointment problem described previously, the facilitator could ask the development team and users to think about their experiences of making appointments and to identify candidate classes based on their past experiences. Notice that this approach does not use the functional models developed earlier. It simply asks the participants to identify the objects with which they have interacted. For example, a potential set of objects that come to mind are doctors, nurses, receptionists, appointment, illness, treatment, prescriptions, insurance card, and medical records. Once a sufficient number of candidate objects have been identified, the participants should discuss and select which of the candidate objects should be considered further. Once these have been identified, further brainstorming can take place to identify potential attributes, operations, and relationships for each of the identified objects.

Bellin and Simone[2] have suggested a set of useful principles to guide a brainstorming session. First, all suggestions should be taken seriously. At this point in the development

[2] D. Bellin and S. S. Simone, *The CRC Card Book* (Reading, MA: Addison-Wesley, 1997).



Use Case Name: Make Old Patient Appt	ID: 2	Importance Level: Low
Primary Actor: Old Patient	Use Case Type: Detail, Essential	

Stakeholders and Interests:
Old Patient – wants to make, change, or cancel an appointment
Doctor – wants to ensure patient's needs are met in a timely manner

Brief Description: This use case describes how we make an appointment as well as changing or canceling an appointment for a previously seen patient.

Trigger: Patient calls and asks for a new appointment or asks to cancel or change an existing appointment.

Type: External

Relationships:

Association:	Old Patient
Include:	
Extend:	Update Patient Information
Generalization:	Manage Appointments

Normal Flow of Events:

1. The Patient contacts the office regarding an appointment.
2. The Patient provides the Receptionist with his or her name and address.
3. If the Patient's information has changed
   Execute the Update Patient Information use case.
4. If the Patient's payment arrangements has changed
   Execute the Make Payments Arrangements use case.
5. The Receptionist asks Patient if he or she would like to make a new appointment, cancel an existing appointment, or change an existing appointment.
   If the patient wants to make a new appointment,
   the S-1: new appointment subflow is performed.
   If the patient wants to cancel an existing appointment,
   the S-2: cancel appointment subflow is performed.
   If the patient wants to change an existing appointment,
   the S-3: change appointment subflow is performed.
6. The Receptionist provides the results of the transaction to the Patient.

SubFlows:

S-1: New Appointment
1. The Receptionist asks the Patient for possible appointment times.
2. The Receptionist matches the Patient's desired appointment times with available dates and times and schedules the new appointment.

S-2: Cancel Appointment
1. The Receptionist asks the Patient for the old appointment time.
2. The Receptionist finds the current appointment in the appointment file and cancels it.

S-3: Change Appointment
1. The Receptionist performs the S-2: cancel appointment subflow.
2. The Receptionist performs the S-1: new appointment subflow.

Alternate/Exceptional Flows:

S-1, 2a1: The Receptionist proposes some alternative appointment times based on what is available in the appointment schedule.

S-1, 2a2: The Patient chooses one of the proposed times or decides not to make an appointment.

TEMPLATE
can be found at www.wiley.com /college/dennis

FIGURE 5-2 Use-Case Description (Figure 4-13)

of the system, it is much better to have to delete something later than to accidentally leave something critical out. Second, all participants should begin thinking fast and furiously. After all ideas are out on the proverbial table, then the participants can be encouraged to ponder the candidate classes they have identified. Third, the facilitator must manage the fast and furious thinking process. Otherwise, the process will be chaotic. Furthermore, the facilitator should ensure that all participants are involved and that a few participants do not dominate the process. To get the most complete view of the problem, we suggest using a round-robin approach wherein participants take turns suggesting candidate classes. Another approach is to use an electronic brainstorming tool that supports anonymity.[3] Fourth, the facilitator can use humor to break the ice so that all participants can feel comfortable in making suggestions.

Common Object Lists

As its name implies, a *common object list* is simply a list of objects common to the business domain of the system. Several categories of objects have been found to help the analyst in creating the list, such as physical or tangible things, incidents, roles, and interactions.[4] Analysts should first look for physical, or *tangible, things* in the business domain. These could include books, desks, chairs, and office equipment. Normally, these types of objects are the easiest to identify. *Incidents* are events that occur in the business domain, such as meetings, flights, performances, or accidents. Reviewing the use cases can readily identify the *roles* that the people play in the problem, such as doctor, nurse, patient, or receptionist. Typically, an *interaction* is a transaction that takes place in the business domain, such as a sales transaction. Other types of objects that can be identified include places, containers, organizations, business records, catalogs, and policies. In rare cases, processes themselves may need information stored about them. In these cases, processes may need an object, in addition to a use case, to represent them. Finally, there are libraries of reusable objects that have been created for different business domains. For example, with regard to the appointment problem, the Common Open Source Medical Objects[5] could be useful to investigate for potential objects that should be included.

Patterns

The idea of using patterns is a relatively new area in object-oriented systems development.[6] There have been many definitions of exactly what a pattern is. From our perspective, a *pattern* is simply a useful group of collaborating classes that provide a solution to a commonly occurring problem. Because patterns provide a solution to commonly occurring problems, they are reusable.

[3] A.R. Dennis, J.S. Valacich, T. Connolly, and B.E. Wynne, "Process Structuring in Electronic Brainstorming," *Information Systems Research* 7, no. 2 (June 1996): 268–277.

[4] For example, see C. Larman, *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design* (Englewood Cliffs, NJ: Prentice Hall, 1998); S. Shlaer and S. J. Mellor, *Object-Oriented Systems Analysis: Modeling the World in Data* (Englewood Cliffs, NJ: Yourdon Press, 1988).

[5] See Common Open Source Medical Objects, Sourceforge, sourceforge.net/projects/cosmos/.

[6] Many books have been devoted to this topic. For example, see P. Coad, D. North, and M. Mayfield, *Object Models: Strategies, Patterns, & Applications,* 2nd Ed. (Englewood Cliffs, NJ: Prentice Hall, 1997); H.-E. Eriksson and M. Penker, *Business Modeling with UML: Business Patterns at Work* (New York: Wiley, 2000); M. Fowler, *Analysis Patterns: Reusable Object Models* (Reading, MA: Addison-Wesley, 1997); E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software* (Reading, MA: Addison-Wesley, 1995); David C. Hay, *Data Model Patterns: Conventions of Thought* (New York: Dorset House, 1996); L. Silverston, *The Data Model Resource Book: A Library of Universal Data Models for All Enterprises, Volume 1, Revised Ed.* (New York, NY; Wiley, 2001).

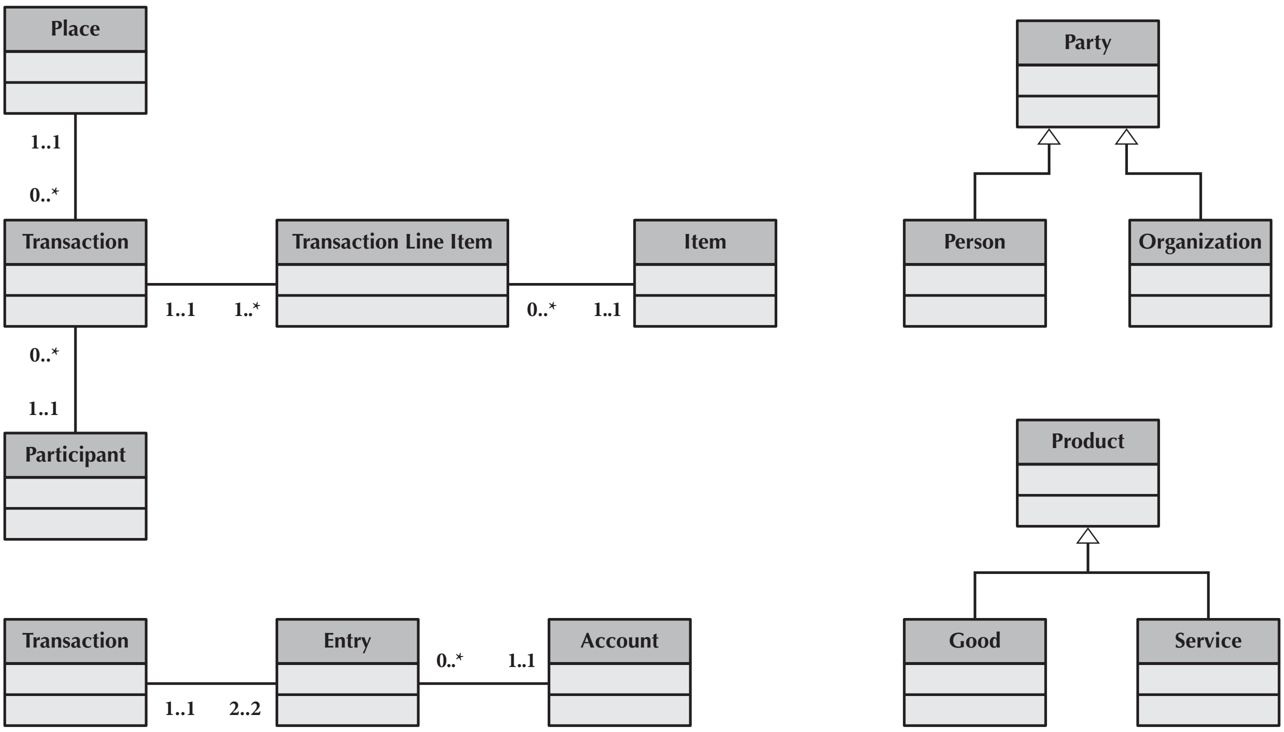


FIGURE 5-3 Sample Patterns

An architect, Christopher Alexander, has inspired much of the work associated with using patterns in object-oriented systems development. According to Alexander and his colleagues,[7] it is possible to make very sophisticated buildings by stringing together commonly found patterns, rather than creating entirely new concepts and designs. In a similar manner, it is possible to put together commonly found object-oriented patterns to form elegant object-oriented information systems. For example, many business transactions involve the same types of objects and interactions. Virtually all transactions would require a transaction class, a transaction line item class, an item class, a location class, and a participant class. By reusing these existing patterns of classes, we can more quickly and more completely define the system than if we start with a blank piece of paper.

Many types of patterns have been proposed, ranging from high-level business-oriented patterns to more low-level design patterns. For example, Figure 5-3 depicts a set of useful analysis patterns.[8] Figure 5-4 portrays a class diagram that we created by merging the patterns contained in Figure 5-3 into a single reusable pattern. In this case, we merged the Transaction–Entry–Account pattern (located at the bottom left of Figure 5-3) with the Place–Transaction–Participant–Transaction Line Item–Item pattern (located at the top left of Figure 5-3) on the

[7] C. Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-King, and S. Angel, *A Pattern Language* (New York: Oxford University Press, 1977).

[8] The patterns are portrayed using UML Class Diagrams. We describe the syntax of the diagrams later in this chapter. The specific patterns shown have been adapted from patterns described in P. Coad, D. North, and M. Mayfield, *Object Models: Strategies, Patterns, & Applications*, 2nd Ed.; M. Fowler, *Analysis Patterns: Reusable Object Models;* L. Silverston, *The Data Model Resource Book: A Library of Universal Data Models for All Enterprises, Volume 1, Revised Edition.*

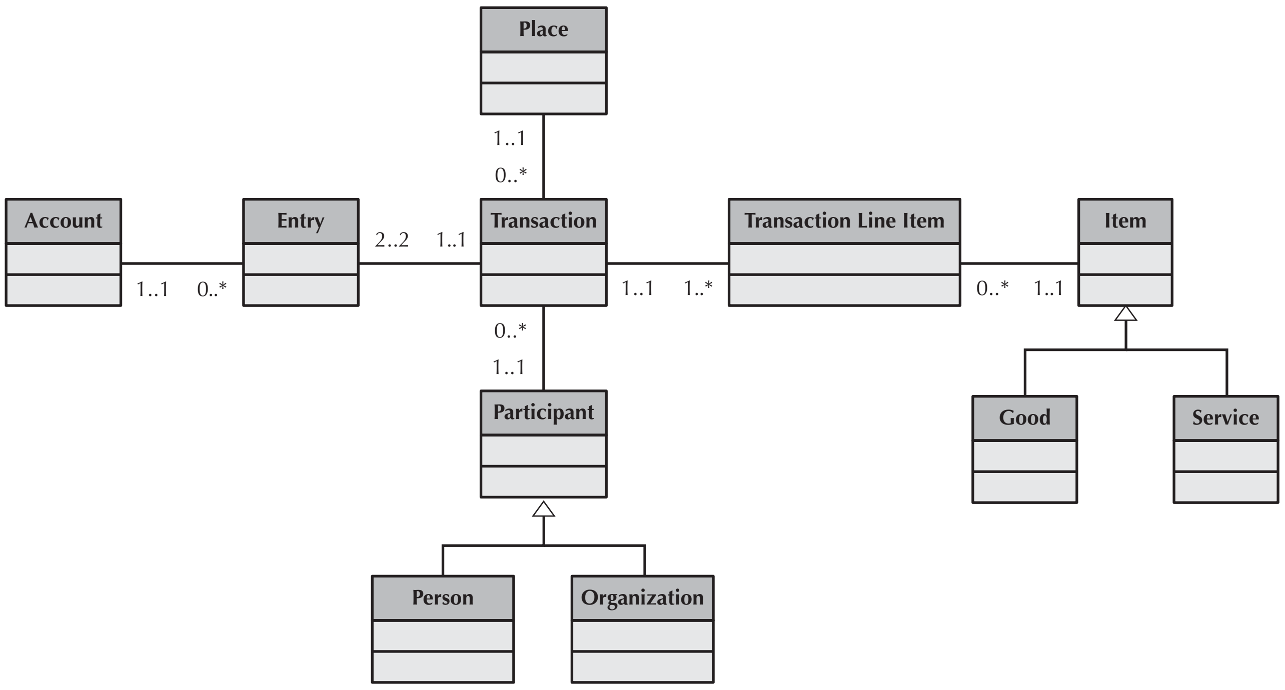


FIGURE 5-4 Sample Integration of Sample Patterns

common Transaction class. Next, we merged the Party–Person–Organization (located at the top right of Figure 5-3) by merging the Participant and Party classes. Finally, we extended the Item class by merging the Item class with the Product class of the Product–Good–Service pattern (located at the bottom right of Figure 5-3).

In this manner, using patterns from different sources in enables the development team to leverage knowledge beyond that of the immediate team members and allows the team to develop more complete and robust models of the problem domain. For example, in the case of the appointment problem, we can look at the objects previously identified through textual analysis, brainstorming, and/or common object lists and see if it makes sense to map any of them into any predefined reusable patterns. In this specific case, we can look at an appointment as a type of transaction in which a doctor's office participates. By looking at an appointment as a type of transaction, we can apply the pattern we created in Figure 5-4 and discover a set of previously unidentified objects, such as Place, Patient as a type of Participant, and Transaction Line Items that are associated with different types of Items (Goods and/or Services). Discovering these specific additional objects could be useful in developing the billing side of the appointment system. Even though these additional objects could be applicable, they were not uncovered using the other techniques.

Based on this simple example, it is obvious that using patterns to develop structural models can be advantageous. Figure 5-5 lists some common business domains for which patterns have been developed and their source. If we are developing a business information system in one of these business domains, then the patterns developed for that domain may be a very useful starting point in identifying needed classes and their attributes, operations, and relationships.

Business Domains	Sources of Patterns
Accounting	3, 4
Actor-Role	2
Assembly-Part	1
Container-Content	1
Contract	2, 4
Document	2, 4
Employment	2, 4
Financial Derivative Contracts	3
Geographic Location	2, 4
Group-Member	1
Interaction	1
Material Requirements Planning	4
Organization and Party	2, 3
Plan	1, 3
Process Manufacturing	4
Trading	3
Transactions	1, 4

1. Peter Coad, David North, and Mark Mayfield, *Object Models: Strategies, Patterns, and Applications,* 2nd Ed. (Englewood Cliffs, NJ: Prentice Hall, 1997).
2. Hans-Erik Eriksson and Magnus Penker, *Business Modeling with UML: Business Patterns at Work* (New York: Wiley, 2000).
3. Martin Fowler, *Analysis Patterns: Reusable Object Models* (Reading, MA: Addison-Wesley, 1997).
4. David C. Hay, *Data Model Patterns: Conventions of Thought* (New York, NY, Dorset House, 1996).

FIGURE 5-5
Useful Patterns

CRC CARDS

CRC (Class–Responsibility–Collaboration) cards are used to document the responsibilities and collaborations of a class. In some object-oriented systems-development methodologies, CRC cards are seen to be an alternative competitor to the Unified Process employment of use cases and class diagrams. However, we see them as a useful, low-tech approach that can compliment a typical high-tech Unified Process approach that uses CASE tools. We use an extended form of the CRC card to capture all relevant information associated with a class.[9] We describe the elements of our CRC cards later, after we explain responsibilities and collaborations.

Responsibilities and Collaborations

Responsibilities of a class can be broken into two separate types: knowing and doing. *Knowing responsibilities* are those things that an instance of a class must be capable of knowing. An instance of a class typically knows the values of its attributes and its relationships. *Doing responsibilities* are those things that an instance of a class must be capable of doing. In this case, an instance of a class can execute its operations or it can request a second instance, which it knows about, to execute one of its operations on behalf of the first instance.

The structural model describes the objects necessary to support the business processes modeled by the use cases. Most use cases involve a set of several classes, not just one class.

[9] Our CRC cards are based on the work of D. Bellin and S. S. Simone, *The CRC Card Book* (Reading, MA: Addison-Wesley, 1997); I. Graham, *Migrating to Object Technology* (Wokingham, England: Addison-Wesley, 1995); B. Henderson-Sellers and B. Unhelkar, *OPEN modeling with UML* (Harlow, England: Addison-Wesley, 2000).

These classes form *collaborations*. Collaborations allow the analyst to think in terms of clients, servers, and contracts.[10] A *client* object is an instance of a class that sends a request to an instance of another class for an operation to be executed. A *server* object is the instance that receives the request from the client object. A *contract* formalizes the interactions between the client and server objects. Chapter 8 provides a more-detailed explanation of contracts and examples of their use.

An analyst can use the idea of class responsibilities and client–server–contract collaborations to help identify the classes, along with the attributes, operations, and relationships, involved with a use case. One of the easiest ways to use CRC cards in developing a structural model is through anthropomorphism—pretending that the classes have human characteristics. Members of the development team can either ask questions of themselves or be asked questions by other members of the team. Typically the questions asked are of the form:

Who or what are you?
What do you know?
What can you do?

The answers to the questions are then used to add detail to the evolving CRC cards. For example, in the appointment problem, a member of the team can pretend that he or she is an appointment. In this case, the appointment would answer that he or she knows about the doctor and patient who participate in the appointment and they would know the date and time of the appointment. Furthermore, an appointment would have to know how to create itself, delete itself, and to possibly change different aspects of itself. In some cases, this approach will uncover additional objects that have to be added to the evolving structural model.

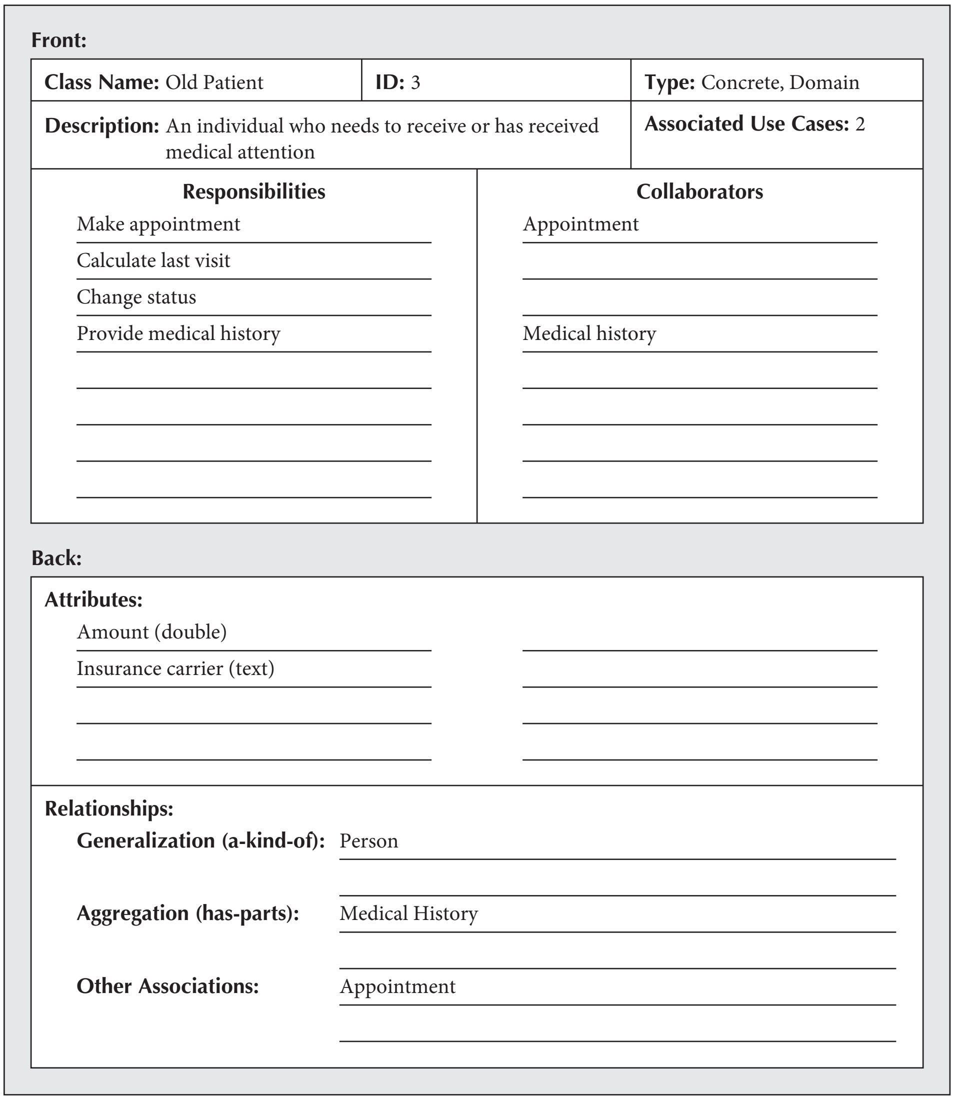
Elements of a CRC Card

The set of CRC cards contains all the information necessary to build a logical structural model of the problem under investigation. Figure 5-6 shows a sample CRC card. Each CRC card captures and describes the essential elements of a class. The front of the card contains the class's name, ID, type, description, associated use cases, responsibilities, and collaborators. The name of a class should be a noun (but not a proper noun, such as the name of a specific person or thing). Just like the use cases, in later stages of development, it is important to be able to trace back design decisions to specific requirements. In conjunction with the list of associated use cases, the ID number for each class can be used to accomplish this. The description is simply a brief statement that can be used as a textual definition for the class. The responsibilities of the class tend to be the operations that the class must contain (i.e., the doing responsibilities).

The back of a CRC card contains the attributes and relationships of the class. The attributes of the class represent the knowing responsibilities that each instance of the class has to meet. Typically, the data type of each attribute is listed with the name of the attribute (e.g., the amount attribute is double and the insurance carrier is text). Three types of relationships typically are captured at this point: generalization, aggregation, and other associations. In Figure 5-6, we see that a Patient is a-kind-of Person and that a Patient is associated with Appointments.

CRC cards are used to document the essential properties of a class. However, once the cards are filled out, the analyst can use the cards and anthropomorphisms in role-playing (described in the next section) to uncover missing properties by executing the different

[10] For more information, see K. Beck and W. Cunningham, "A Laboratory for Teaching Object-Oriented Thinking," *Proceedings of OOPSLA, SIGPLAN Notices,* 24, no. 10 (1989): 1–6; B. Henderson-Sellers and B. Unhelkar, *OPEN Modeling with UML* (Harlow, England: Addison-Wesley, 2000); C. Larman, *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design* (Englewood Cliffs, NJ: Prentice Hall, 1998); B. Meyer, *Object-Oriented Software Construction* (Englewood Cliffs, NJ: Prentice Hall, 1994); R. Wirfs-Brock, B. Wilkerson, and L. Wiener, *Designing Object-Oriented Software* (Englewood Cliffs, NJ, Prentice Hall, 1990).



Front:

Class Name: Old Patient	**ID:** 3	**Type:** Concrete, Domain
Description: An individual who needs to receive or has received medical attention		**Associated Use Cases:** 2

Responsibilities	**Collaborators**
Make appointment	Appointment
Calculate last visit	
Change status	
Provide medical history	Medical history

Back:

Attributes:

Amount (double)

Insurance carrier (text)

Relationships:

Generalization (a-kind-of):	Person
Aggregation (has-parts):	Medical History
Other Associations:	Appointment

FIGURE 5-6
Sample CRC Card

scenarios associated with the use cases (see Chapter 4). Role-playing also can be used as a basis to test the clarity and completeness of the evolving representation of the system.

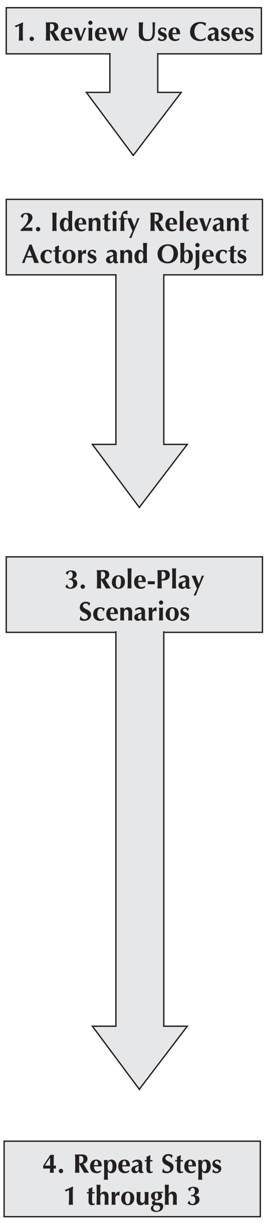
Role-Playing CRC Cards with Use Cases[11, 12]

In addition to the object identification approaches described earlier (textual analysis, brainstorming, common object lists, and patterns), CRC cards can be used in a *role-playing* exercise that has been shown to be useful in discovering additional objects, attributes, relationships, and operations. Furthermore, in addition to walkthroughs, described later in this chapter, role-playing is very useful in testing the fidelity of the evolving structural model. In general, members of the team perform roles associated with the actors and objects previously identified

[11] This step is related to the verification and validation of the analysis models (functional, structural, and behavioral). Because this deals with verification and validation that take place between the models, in this case functional and structural, we will return to this topic in Chapter 7.

[12] Our role-playing approach is based on the work of D. Bellin and S. S. Simone, *The CRC Card Book* (Reading, MA: Addison-Wesley, 1997).

with the different use cases. Technically speaking, the members of the team perform the different steps associated with a specific scenario of a use case. Remember, a scenario is a single, unique execution path through a use case. A useful place to look for the different scenarios of a use case is the activity diagrams (e.g., see Figures 4-8, 4-9, 4-10, and 4-12). A different scenario exists for each time a decision node causes a split in the execution path of the use case. Also, scenarios can be identified from the alternative/exceptional flows in a use-case description. Considering the incremental and iterative nature and that activity diagrams and use-case descriptions should contain the same information, reviewing both representations will ensure that relevant scenarios are not missed.


1. Review Use Cases
2. Identify Relevant Actors and Objects
3. Role-Play Scenarios
4. Repeat Steps 1 through 3

The first step is to review the use-case descriptions (see Figure 5-2). This allows the team to pick a specific use case to role-play. Even though it is tempting to try to complete as many use cases as possible in a short time, the team should not choose the easiest use cases first. Instead, at this point in the development of the system, the team should choose the use case that is the most important, the most complex, or the least understood.

The second step is to identify the relevant roles that are to be played. Each role is associated with either an actor or an object. To choose the relevant objects, the team reviews each of the CRC cards and picks the ones that are associated with the chosen use case. For example, in Figure 5-6, we see that the CRC card that represents the Old Patient class is associated with Use Case number 2. So if we were going to role-play the Make Old Patient Appt use case (see Figure 5-2), we would need to include the Old Patient CRC card. By reviewing the use-case description, we can easily identify the Old Patient and Doctor actors (see Primary Actor and Stakeholders section of the use case description in Figure 5-2). By reading the event section of the use-case description, we identify the internal actor role of Receptionist. After identifying all of the relevant roles, we assign each one to a different member of the team.

The third step is to role-play scenarios of the use case by having the team members perform each one. To do this, each team member must pretend that he or she is an instance of the role assigned to him or her. For example, if a team member was assigned the role of the Receptionist, then he or she would have to be able to perform the different steps in the scenario associated with the Receptionist. In the case of the change appointment scenario, this would include steps 2, 5, 6, S-3, S-1, and S-2. However, when this scenario is performed (role-played), it would be discovered that steps 1, 3, and 4 were incomplete. For example, in Step 1, what actually occurs? Does the Patient make a phone call? If so, who answers the phone? In other words, a lot of information contained in the use-case description is only identified in an implicit, not explicit, manner. When the information is not identified explicitly, there is a lot of room for interpretation, which requires the team members to make assumptions. It is much better to remove the need to make an assumption by making each step explicit. In this case, Step 1 of the Normal Flow of Events should be modified. Once the step has been fixed, the scenario is tried again. This process is repeated until the scenario can be executed to a successful conclusion. Once the scenario has successfully concluded, the next scenario is performed. This is repeated until all of the scenarios of the use case can be performed successfully. [13]

The fourth step is to simply repeat steps 1 through 3 for the remaining use cases.

[13] In some cases, some scenarios are only executed in very rare circumstances. So, from a practical perspective, each scenario could be prioritized individually and only "important" scenarios would have to be implemented for the first release of the system. Only those scenarios would have to be tested at this point in the evolution of the system.

CLASS DIAGRAMS

A *class diagram* is a *static model* that shows the classes and the relationships among classes that remain constant in the system over time. The class diagram depicts classes, which include both behaviors and states, with the relationships between the classes. The following sections present the elements of the class diagram, different approaches that can be used to simplify a class diagram, and an alternative structure diagram: the object diagram.

Elements of a Class Diagram

Figure 5-7 shows a class diagram that was created to reflect the classes and relationships associated with the appointment system. This diagram is based on the classes uncovered through the object identification techniques and the role-playing of the CRC cards described earlier.

Class The main building block of a class diagram is the class, which stores and manages information in the system (see Figure 5-8). During analysis, classes refer to the people, places, and things about which the system will capture information. Later, during design and implementation, classes can refer to implementation-specific artifacts such as windows, forms, and other objects used to build the system. Each class is drawn using a three-part rectangle, with the class's name at the top, attributes in the middle, and operations at the bottom. We can see that the classes identified earlier, such as Participant, Doctor, Patient, Receptionist, Medical History, Appointment, and Symptom, are included in Figure 5-7. The attributes of a class and their values define the *state* of each object created from the class, and the behavior is represented by the operations.

Attributes are properties of the class about which we want to capture information (see Figure 5-8). Notice that the Participant class in Figure 5-7 contains the attributes: lastname, firstname, address, phone, and birthdate. At times, you might want to store *derived attributes*, which are attributes that can be calculated or derived; these special attributes are denoted by placing a slash (/) before the attribute's name. Notice how the person class contains a derived attribute called/age, which can be derived by subtracting the patient's birth date from the current date. It is also possible to show the *visibility* of the attribute on the diagram. Visibility relates to the level of information hiding to be enforced for the attribute. The visibility of an attribute can be public (+), protected (#), or private (−). A *public* attribute is one that is not hidden from any other object. As such, other objects can modify its value. A *protected* attribute is one that is hidden from all other classes except its immediate subclasses. A *private* attribute is one that is hidden from all other classes. The default visibility for an attribute is normally private.

Operations are actions or functions that a class can perform (see Figure 5-8). The functions that are available to all classes (e.g., create a new instance, return a value for a particular attribute, set a value for a particular attribute, delete an instance) are not explicitly shown within the class rectangle. Instead, only operations unique to the class are included, such as the cancel without notice operation in the Appointment class and the calculate last visit operation in the Patient class in Figure 5-7. Notice that both the operations are followed by parentheses, which contain the parameter(s) needed by the operation. If an operation has no parameters, the parentheses are still shown but are empty. As with attributes, the visibility of an operation can be designated public, protected, or private. The default visibility for an operation is normally public.

There are four kinds of operations that a class can contain: constructor, query, update, and destructor. A *constructor operation* creates a new instance of a class. For example, the patient class may have a method called insert (), which creates a new patient instance as

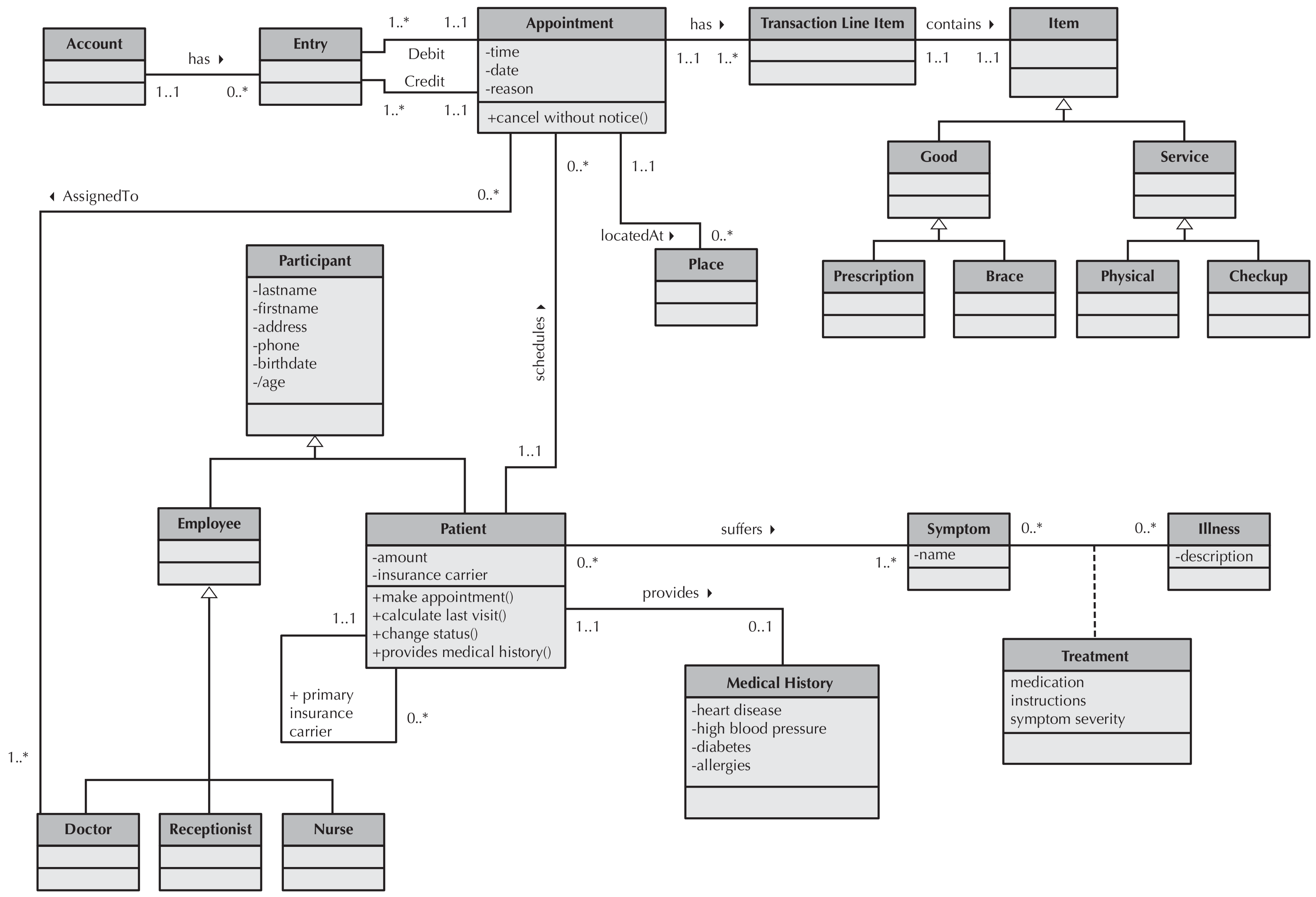


FIGURE 5-7 Sample Class Diagram

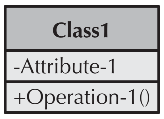
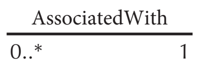
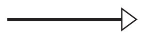
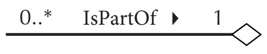
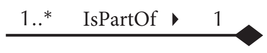
A class: • Represents a kind of person, place, or thing about which the system will need to capture and store information. • Has a name typed in bold and centered in its top compartment. • Has a list of attributes in its middle compartment. • Has a list of operations in its bottom compartment. • Does not explicitly show operations that are available to all classes.	 **Class1** -Attribute-1 +Operation-1()
An attribute: • Represents properties that describe the state of an object. • Can be derived from other attributes, shown by placing a slash before the attribute's name.	attribute name /derived attribute name
An operation: • Represents the actions or functions that a class can perform. • Can be classified as a constructor, query, or update operation. • Includes parentheses that may contain parameters or information needed to perform the operation.	operation name ()
An association: • Represents a relationship between multiple classes or a class and itself. • Is labeled using a verb phrase or a role name, whichever better represents the relationship. • Can exist between one or more classes. • Contains multiplicity symbols, which represent the minimum and maximum times a class instance can be associated with the related class instance.	 AssociatedWith 0..* 1
A generalization: • Represents a-kind-of relationship between multiple classes.	
An aggregation: • Represents a logical a-part-of relationship between multiple classes or a class and itself. • Is a special form of an association.	 0..* IsPartOf ▸ 1
A composition: • Represents a physical a-part-of relationship between multiple classes or a class and itself • Is a special form of an association.	 1..* IsPartOf ▸ 1

FIGURE 5-8
Class Diagram Syntax

patients are entered into the system. As we just mentioned, if an operation implements one of the basic functions (e.g., create a new instance), it is normally not explicitly shown on the class diagram, so typically we do not see constructor methods explicitly on the class diagram.

A *query operation* makes information about the state of an object available to other objects, but it does not alter the object in any way. For instance, the calculate last visit () operation that determines when a patient last visited the doctor's office will result in the object's being accessed by the system, but it will not make any change to its information. If a query method merely asks for information from attributes in the class (e.g., a patient's name, address, phone), then it is not shown on the diagram because we assume that all objects have operations that produce the values of their attributes.

An *update operation* changes the value of some or all the object's attributes, which may result in a change in the object's state. Consider changing the status of a patient from new to current with a method called change status() or associating a patient with a particular appointment with make appointment (appointment). If the result of the operation can change the state of the object, then the operation must be explicitly included on the class diagram. On the other hand, if the update operation is a simple assignment operation, it can be omitted from the diagram.

A *destructor operation* simply deletes or removes the object from the system. For example, if an employee object no longer represents an actual employee associated with the firm, the employee could need to be removed from the employee database, and a destructor operation would be used to implement this behavior. However, deleting an object is one of the basic functions and therefore would not be included on the class diagram.

Relationships A primary purpose of a class diagram is to show the relationships, or associations, that classes have with one another. These are depicted on the diagram by drawing lines between classes (see Figure 5-8). When multiple classes share a relationship (or a class shares a relationship with itself), a line is drawn and labeled with either the name of the relationship or the roles that the classes play in the relationship. For example, in Figure 5-7 the two classes patient and appointment are associated with one another whenever a patient schedules an appointment. Thus, a line labeled schedules connects patient and appointment, representing exactly how the two classes are related to each other. Also, notice that there is a small solid triangle beside the name of the relationship. The triangle allows a direction to be associated with the name of the relationship. In Figure 5-7, the schedules relationship includes a triangle, indicating that the relationship is to be read as "patient schedules appointment." Inclusion of the triangle simply increases the readability of the diagram. In Figure 5-9, three additional examples of associations are portrayed: An Invoice is AssociatedWith a Purchase Order (and vice versa), a Pilot Flies an Aircraft, and a Spare Tire IsLocatedIn a Trunk.

Sometimes a class is related to itself, as in the case of a patient being the primary insurance carrier for other patients (e.g., spouse, children). In Figure 5-7, notice that a line was drawn between the patient class and itself and called *primary insurance carrier* to depict the role that the class plays in the relationship. Notice that a plus (+) sign is placed before the label to communicate that it is a role as opposed to the name of the relationship. When labeling an association, we use either a relationship name or a role name (not both), whichever communicates a more thorough understanding of the model.

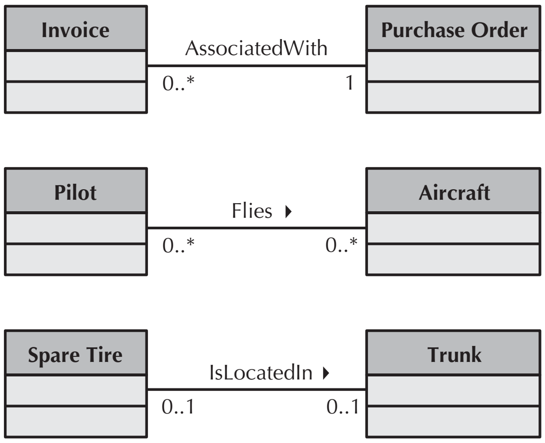
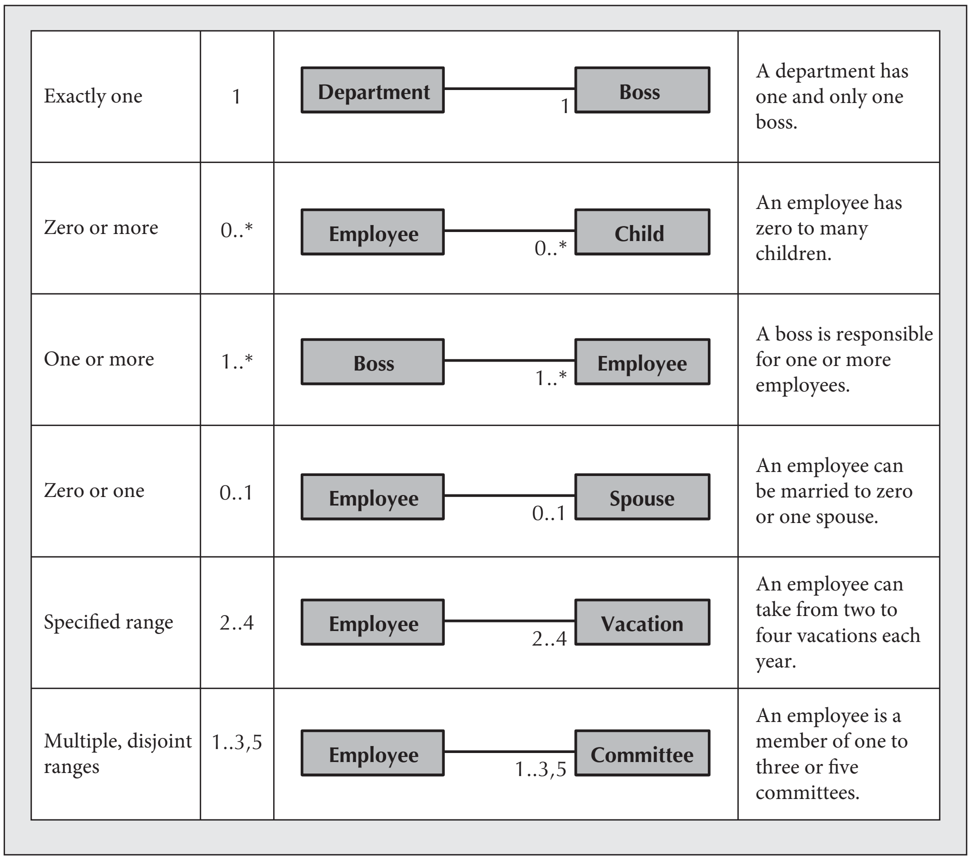


FIGURE 5-9
Sample Association



Exactly one	1	Department 1 Boss	A department has one and only one boss.
Zero or more	0..*	Employee 0..* Child	An employee has zero to many children.
One or more	1..*	Boss 1..* Employee	A boss is responsible for one or more employees.
Zero or one	0..1	Employee 0..1 Spouse	An employee can be married to zero or one spouse.
Specified range	2..4	Employee 2..4 Vacation	An employee can take from two to four vacations each year.
Multiple, disjoint ranges	1..3,5	Employee 1..3,5 Committee	An employee is a member of one to three or five committees.

FIGURE 5-10
Multiplicity

Relationships also have *multiplicity*, which documents how an instance of an object can be associated with other instances. Numbers are placed on the association path to denote the minimum and maximum instances that can be related through the association in the format minimum number.. maximum number (see Figure 5-10). The numbers specify the relationship from the class at the far end of the relationship line to the end with the number. For example, in Figure 5-7, there is a 0..* on the appointment end of the patient schedules appointment relationship. This means that a patient can be associated with zero through many different appointments. At the patient end of this same relationship, there is a 1..1, meaning that an appointment must be associated with one and only one patient. In Figure 5-9, we see that an instance of the Invoice class must be AssociatedWith one instance of the Purchase Order class and that an instance of the Purchase Order class may be AssociatedWith zero or more instances of the Invoice class, that an instance of the Pilot class Flies zero or more instances of the Aircraft class, and that an instance of the Aircraft class may be flown by zero or more instances of the Pilot class. Finally, we see that an instance the Spare Tire class IsLocatedIn zero or one instance of the Trunk class, whereas an instance of the Trunk class can contain zero or one instance of the Spare Tire class.

There are times when a relationship itself has associated properties, especially when its classes share a many-to-many relationship. In these cases, a class called an *association class* is formed, which has its own attributes and operations.[14] It is shown as a rectangle attached

[14] For those familiar with data modeling, associative classes serve a purpose similar to the one the associative entity serves in ER diagramming.

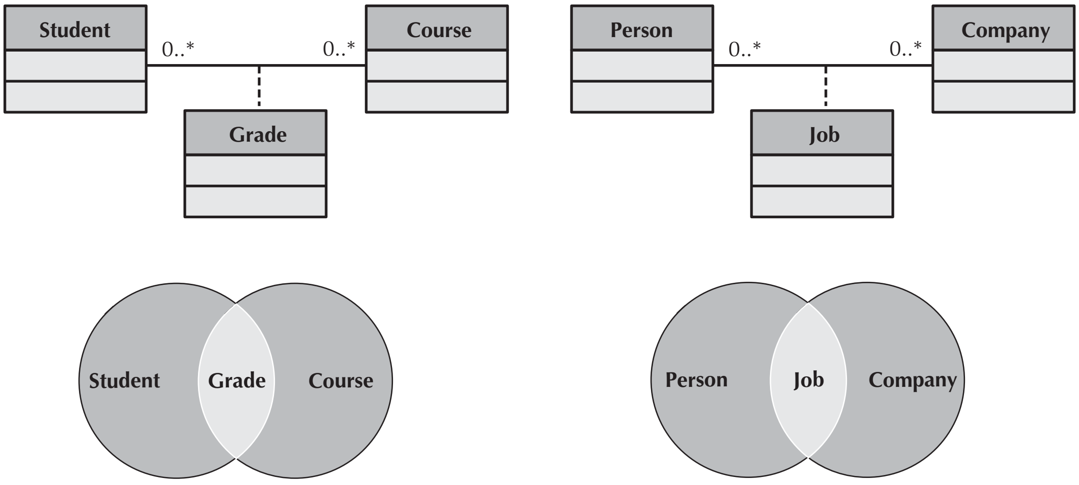


FIGURE 5-11 Sample Association Classes

by a dashed line to the association path, and the rectangle's name matches the label of the association. Think about the case of capturing information about illnesses and symptoms. An illness (e.g., the flu) can be associated with many symptoms (e.g., sore throat, fever), and a symptom (e.g., sore throat) can be associated with many illnesses (e.g., the flu, strep throat, the common cold). Figure 5-7 shows how an association class can capture information about remedies that change depending on the various combinations. For example, a sore throat caused by strep throat requires antibiotics, whereas treatment for a sore throat from the flu or a cold could be throat lozenges or hot tea. Another way to decide when to use an association class is when attributes that belong to the intersection of the two classes involved in the association must be captured. We can visually think about an association class as a Venn diagram. For example, in Figure 5-11, the Grade idea is really an intersection of the Student and Course classes, because a grade exists only at the intersection of these two ideas. Another example shown in Figure 5-11 is that a job may be viewed as the intersection between a Person and a Company. Most often, classes are related through a normal association; however, there are two special cases of an association that you will see appear quite often: generalization and aggregation.

Generalization and Aggregation Associations A *generalization association* shows that one class (subclass) inherits from another class (superclass), meaning that the properties and operations of the superclass are also valid for objects of the subclass. The generalization path is shown with a solid line from the subclass to the superclass and a hollow arrow pointing at the superclass (see Figure 5-8). For example, Figure 5-7 communicates that doctors, nurses, and receptionists are all kinds of employees and those employees and patients are kinds of participants. Remember that the generalization relationship occurs when you need to use words like "is a kind of" to describe the relationship. Some additional examples of generalization are given in Figure 5-12. For example, Cardinal is a-kind-of Bird, which is a-kind-of Animal; a General Practitioner is a-kind-of Physician, which is a-kind-of Person; and a Truck is a-kind-of Land Vehicle, which is a-kind-of Vehicle.

An *aggregation association* is used when classes actually comprise other classes. For example, think about a doctor's office that has decided to create health care teams that include doctors, nurses, and administrative personnel. As patients enter the office, they are assigned to a health care team, which cares for their needs during their visits. We could

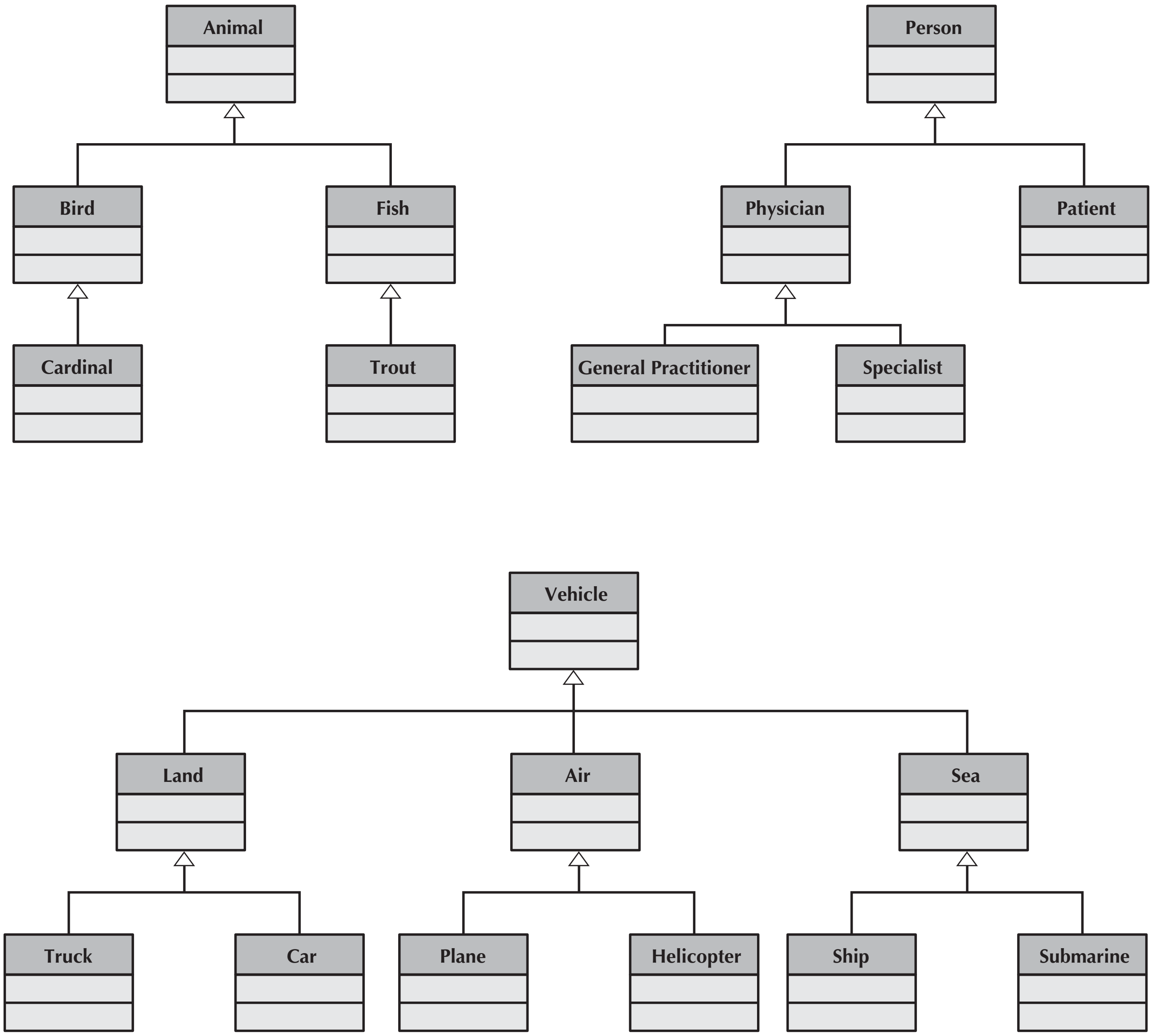


FIGURE 5-12 Sample Generalizations

include this new knowledge in Figure 5-7 by adding two new classes (Administrative Personnel and Health Team) and aggregation relationships from the Doctor, the Nurse, and the new Administrative Personnel classes to the new Health Team class. A diamond is placed nearest the class representing the aggregation (health care team), and lines are drawn from the diamond to connect the classes that serve as its parts (doctors, nurses, and administrative personnel). Typically, you can identify these kinds of associations when you need to use words like "is a part of" or "is made up of" to describe the relationship. However, from a UML perspective, there are two types of aggregation associations: aggregation and composition (see Figure 5-8).

Aggregation is used to portray logical a-part-of relationships and is depicted on a UML class diagram by a hollow or white diamond. For example in Figure 5-13, three logical

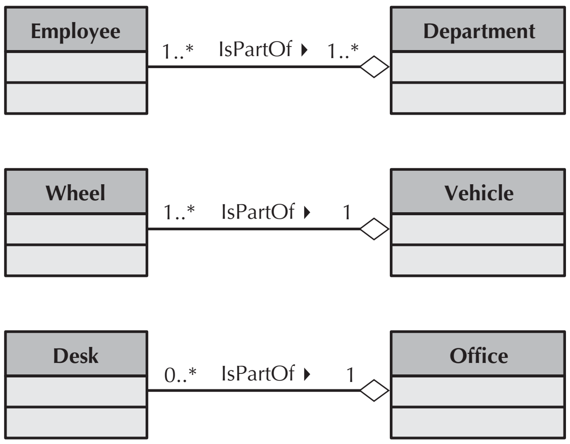


FIGURE 5-13
Sample Aggregation Associations

aggregations are shown. Logical implies that it is possible for a part to be associated with multiple wholes or that is relatively simple for the part to be removed from the whole. For example, an instance of the Employee class IsPartOf an instance of at least one instance of the Department class, an instance of the Wheel class IsPartOf an instance of the Vehicle class, and an instance of the Desk class IsPartOf an instance of the Office class. Obviously, in many cases an employee can be associated with more than one department, and it is relatively easy to remove a wheel from a vehicle or move a desk from an office.

Composition is used to portray a physical part of relationships and is shown by a black diamond. *Physical* implies that the part can be associated with only a single whole. For example in Figure 5-14, three physical compositions are illustrated: an instance of a door can be a part of only a single instance of a car, an instance of a room can be a part of an instance only of a single building, and an instance of a button can be a part of only a single mouse. However, in many cases, the distinction that you can achieve by including aggregation (white diamonds) and composition (black diamonds) in a class diagram might not be worth the price of adding additional graphical notation for the client to learn. Therefore, many UML experts view the inclusion of aggregation and composition notation to the UML class diagram as simply "syntactic sugar" and not necessary because the same information can always be portrayed by simply using the association syntax.

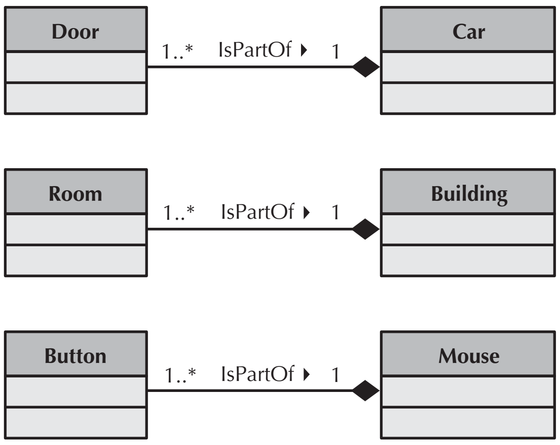


FIGURE 5-14
Sample Composition Associations

Simplifying Class Diagrams

When a class diagram is fully populated with all the classes and relationships for a real-world system, the class diagram can become very difficult to interpret (i.e., can be very complex). When this occurs, it is sometimes necessary to simplify the diagram. One way to simplify the class diagram is to show only concrete classes.[15] However, depending on the number of associations that are connected to abstract classes—and thus inherited down to the concrete classes—this particular suggestion could make the diagram more difficult to comprehend.

A second way to simplify the class diagram is through the use of a *view* mechanism. Views were developed originally with relational database management systems to show only a subset of the information contained in the database. In this case, the view would be a useful subset of the class diagram, such as a use-case view that shows only the classes and relationships relevant to a particular use case. A second view could be to show only a particular type of relationship: aggregation, association, or generalization. A third type of view is to restrict the information shown with each class, for example, show only the name of the class, the name and attributes, or the name and operations. These view mechanisms can be combined to further simplify the diagram.

A third approach to simplifying a class diagram is through the use of *packages* (i.e., logical groups of classes). To make the diagrams easier to read and keep the models at a reasonable level of complexity, the classes can be grouped together into packages. Packages are general constructs that can be applied to any of the elements in UML models. In Chapter 4, we introduced the package idea to simplify use-case diagrams. In the case of class diagrams, it is simple to sort the classes into groups based on the relationships that they share.[16]

Object Diagrams

Although class diagrams are necessary to document the structure of the classes, a second type of *static structure diagram*, called an object diagram, can be useful in revealing additional information. An *object diagram* is essentially an instantiation of all or part of a class diagram. *Instantiation* means to create an instance of the class with a set of appropriate attribute values.

Object diagrams can be very useful when trying to uncover details of a class. Generally speaking, it is easier to think in terms of concrete objects (instances) rather than abstractions of objects (classes). For example in Figure 5-15, a portion of the class diagram in Figure 5-7 has been copied and instantiated. The top part of the figure simply is a copy of a small view of the overall class diagram. The lower portion is the object diagram that instantiates that subset of classes. By reviewing the actual instances involved, John Doe, Appt1, Symptom1, and Dr. Smith, we may discover additional relevant attributes, relationships, and/or operations or possibly misplaced attributes, relationships, and/or operations. For example, an appointment has a reason attribute. Upon closer examination, the reason attribute might have been better modeled as an association with the Symptom class. Currently, the Symptom class is associated with the Patient class. After reviewing the object diagram, this seems to be in error. Therefore, we should modify the class diagram to reflect this new understanding of the problem.

[15] See footnote 1.

[16] For those familiar with structured analysis and design, packages serve a purpose similar to the leveling and balancing processes used in data flow diagramming. Packages and package diagrams are described in more detail in Chapter 7.

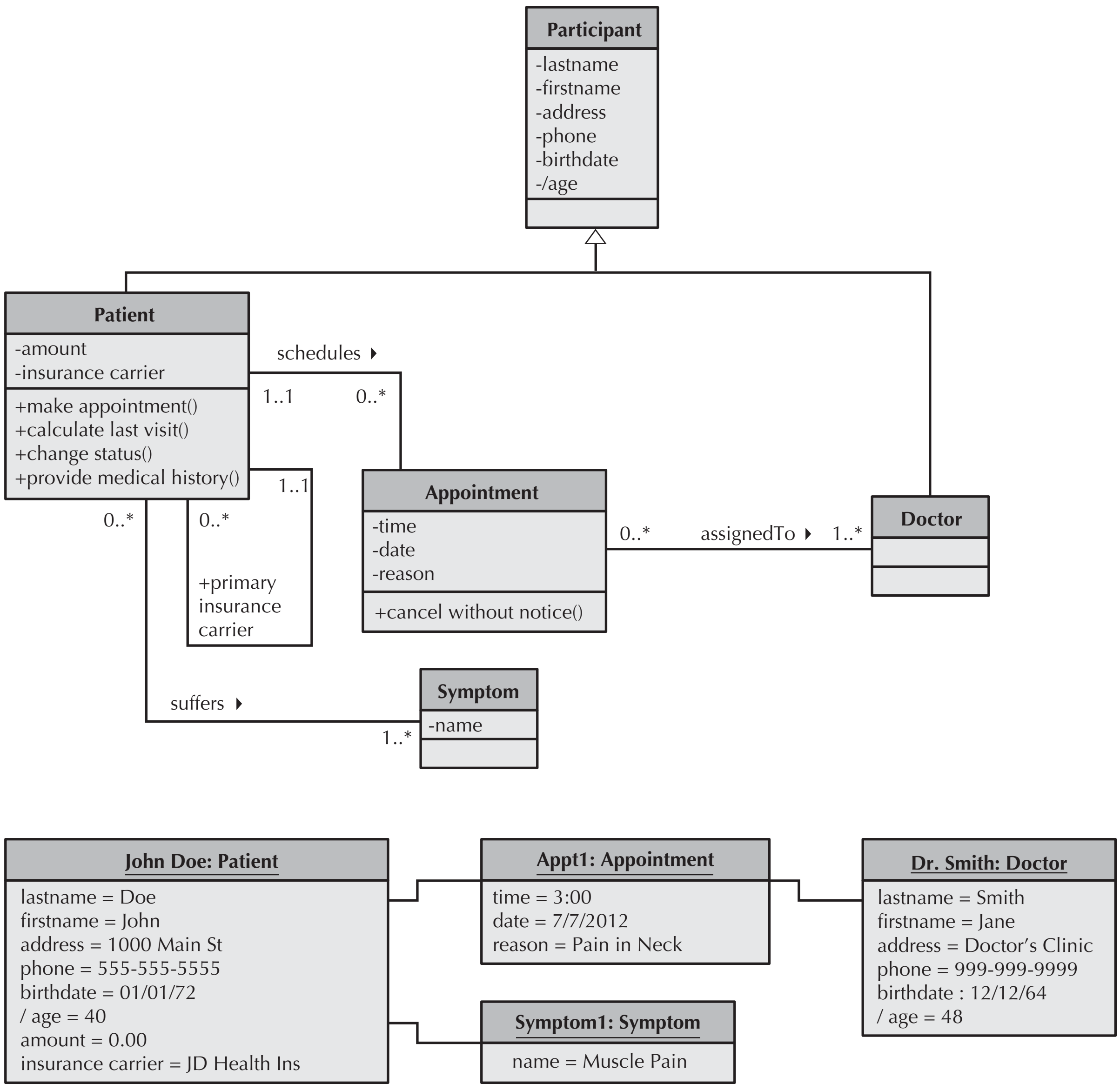
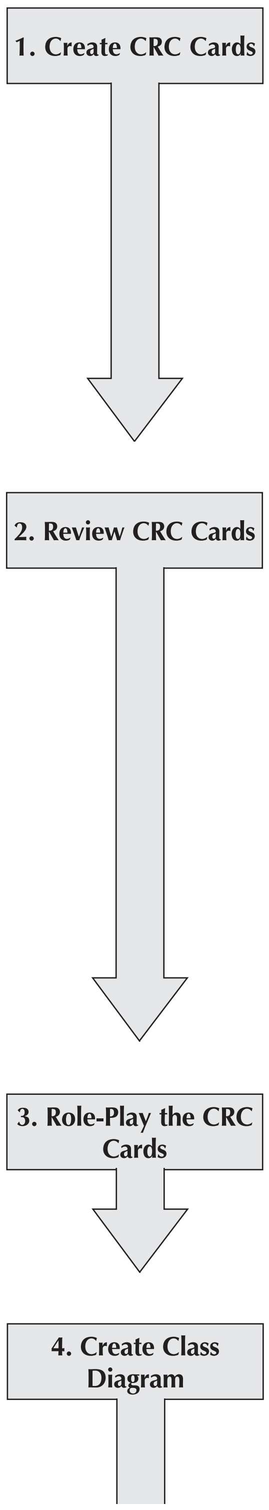


FIGURE 5-15 Sample Object Diagram

CREATING STRUCTURAL MODELS USING CRC CARDS AND CLASS DIAGRAMS

Creating a structural model is an incremental and iterative process whereby the analyst makes a rough cut of the model and then refines it over time. Structural models can become quite complex—in fact, there are systems that have hundreds of classes. It is important to remember that CRC cards and class diagrams can be used to describe both the as-is and to-be structural models of the evolving system, but they are most often used for the to-be model. There are many different ways to identify a set of candidate objects and to create CRC cards and class diagrams. Today most object identification begins with the use cases identified for the problem (see Chapter 4). In this section, we describe a use-case–driven process that can be used to create the structural model of a problem domain.



We could begin creating the structural model with a class diagram instead of CRC cards. However, owing to the low-tech nature and the ease of role-playing use-case scenarios with CRC cards, we prefer to create the CRC cards first and then transfer the information from the CRC cards into a class diagram later. As a result, the first step of our recommended process is to create CRC cards. Performing textual analysis on the use-case descriptions does this. If you recall, the normal flow of events, subflows, and alternative/exceptional flows of the use-case description were written in a special form called Subject–Verb–Direct-Object–Preposition–Indirect object (*SVDPI*). By writing the use-case events in this form, it is easier to use the guidelines for textual analysis in Figure 5-1 to identify the objects. Reviewing the primary actors, stakeholders and interests, and brief descriptions of each use case allows additional candidate objects to be identified. It is useful to go back and review the original requirements to look for information that was not included in the text of the use cases. Record all the uncovered information for each candidate object on a CRC card.

The second step is to review the CRC cards to determine if additional candidate objects, attributes, operations, and relationships are missing. In conjunction with this review, using the brainstorming and common object list approaches described earlier can aid the team in identifying missing classes, attributes, operations, and relationships. For example, the team could start a brainstorming session with a set of questions such as:

- What are the tangible things associated with the problem?
- What are the roles played by the people in the problem domain?
- What incidents and interactions take place in the problem domain?

As you can readily see, by beginning with the use-case descriptions, many of these questions already have partial answers. For example, the primary actors and stakeholders are the roles that are played by the people in the problem domain. However, it is possible to uncover additional roles not thought of previously. This obviously would cause the use-case descriptions, and possibly the use-case diagram, to be modified and possibly expanded. As in the previous step, be sure to record all the uncovered information onto the CRC cards. This includes any modifications uncovered for any previously identified candidate objects and any information regarding any new candidate objects identified.

The third step is to role-play each use-case scenario using the CRC cards. Each CRC card should be assigned to an individual who will perform the operations for the class on the CRC card. As the performers act out their roles, the system tends to break down. When this occurs, additional objects, attributes, operations, or relationships will be identified. Again, as in the previous steps, any time any new information is discovered, new CRC cards are created or modifications to existing CRC cards are made.

The fourth step is to create the class diagram based on the CRC cards. Information contained on the CRC cards is transferred to the class diagrams. The responsibilities are transferred as operations; the attributes are drawn as attributes; and the relationships are drawn as generalization, aggregation, or association relationships. However, the class diagram also requires that the visibility of the attributes and operations be known. As a general rule, attributes are private and operations are public. Therefore, unless the analyst has a good reason to change the default visibility of these properties, then the defaults should be accepted. Finally, the analyst should examine the model for additional opportunities to use aggregation or generalization relationships. These types of relationships can simplify the individual class descriptions. As in the previous steps, all changes must be recorded on the CRC cards.

5. Review Class Diagram

The fifth step is to review the structural model for missing and/or unnecessary classes, attributes, operations, and relationships. Until this step, the focus of the process has been on adding information to the evolving model. At this point, the focus begins to switch from simply adding information to also challenging the reasons for including the information contained in the model. One very useful approach here is to play devil's advocate, where a team member, just for the sake of being a pain in the neck, challenges the reasoning for including all aspects of the model.

6. Incorporate Patterns

The sixth step is to incorporate useful patterns into the evolving structural model. A useful pattern is one that would allow the analyst to more fully describe the underlying domain of the problem being investigated. Looking at the collection of patterns available (Figure 5-5) and comparing the classes contained in the patterns with those in the evolving class diagram enable this. After identifying the useful patterns, the analyst incorporates the identified patterns into the class diagram and modifies the affected CRC cards. This includes adding and removing classes, attributes, operations, and/or relationships.

7. Review the Model

The seventh and final step is to validate the structural model, including both the CRC cards and the class diagram. We discuss this content in the next section of the chapter and in Chapter 7.

Campus Housing Example

In the previous chapter, we identified a set of use cases (Add an Apartment, Delete an Apartment, and Search Available Rental Units) for the campus housing service that helps students find apartments. By reviewing the use cases, we can easily determine that the campus housing service must keep track of information for each available apartment and its owner. The information to be captured for each apartment is the location of the apartment, the number of bedrooms in the apartment, the monthly rent, and how far the apartment is from the campus. Regarding the owner of the apartment, we need to capture the owner's contact information (e.g., name, address, phone number, e-mail address). Since students are simply users of the system, there is no need to capture any information regarding them; that is, in this case, students are simply actors. Finally, with regards to relationships among the classes, there is a single, optional, one to many association relationship that links the two classes together. The Apartment Owner CRC card is portrayed in Figure 5-16, and the class diagram for this situation is shown in Figure 5-17.

Library Example

As with the Campus Housing example, the first step is to create the CRC cards that represent the classes in the structural model. In the previous chapter, we used the Library Book Collection Management System example to describe the process of creating the functional models (use-case and activity diagrams and use-case descriptions). In this chapter, we follow the same familiar example. Because we are following a use-case-driven approach to object-oriented systems development, we first review the events described in the use-case descriptions (see Figure 5-18).

Front:

Class Name: Apartment Owner	**ID:** 1	**Type:** Concrete, Domain
Description: An apartment owner who has apartments for rent		**Associated Use Cases:** 2

Responsibilities	**Collaborators**
Add apartment	Apartment
Delete apartment	Apartment

Back:

Attributes:

Name (string)

Address (address)

Phone number (PhoneNumber)

Email (EmailAddress)

Relationships:

Generalization (a-kind-of):	
Aggregation (has-parts):	
Other Associations:	Apartment

FIGURE 5-16
Campus Housing Apartment Owner CRC Card

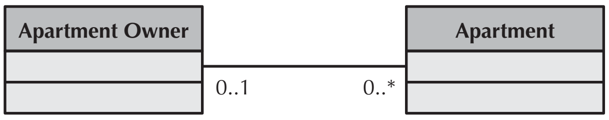
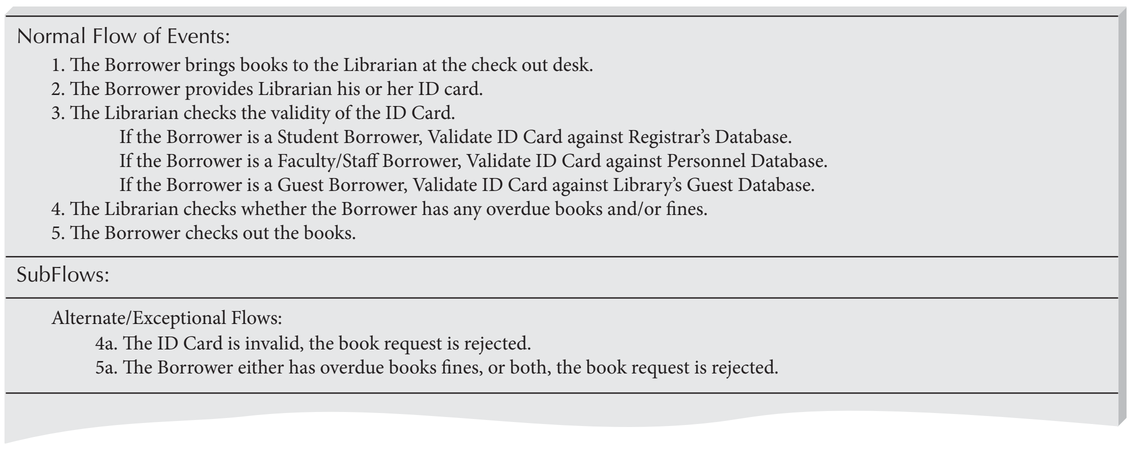


FIGURE 5-17
Campus Housing Class Diagram

Next, we perform textual analysis on the events by applying the textual analysis rules described in Figure 5-1. In this case, we can quickly identify the need to include classes for Borrower, Books, Librarian, Check Out Desk, ID Card, Student Borrower, Faculty/Staff Borrower, Guest Borrower, Registrar's Database, Personnel Database, Library's Guest Database, Overdue Books, Fines, Book Request. We also can easily identify operations to "check the validity" of a book request, to "check out" the books, and to "reject" a book request. Furthermore, the events suggest a "brings" relationship between Borrower and Books and a "provides" relationship between Borrower and Librarian. This step



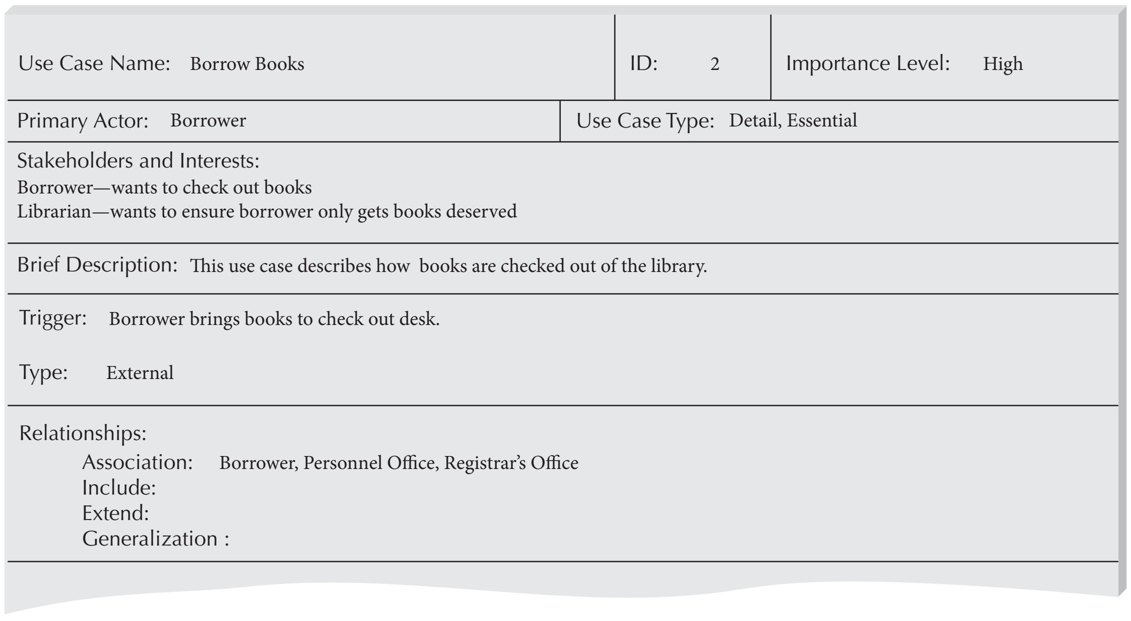
Normal Flow of Events:

1. The Borrower brings books to the Librarian at the check out desk.
2. The Borrower provides Librarian his or her ID card.
3. The Librarian checks the validity of the ID Card.
   - If the Borrower is a Student Borrower, Validate ID Card against Registrar's Database.
   - If the Borrower is a Faculty/Staff Borrower, Validate ID Card against Personnel Database.
   - If the Borrower is a Guest Borrower, Validate ID Card against Library's Guest Database.
4. The Librarian checks whether the Borrower has any overdue books and/or fines.
5. The Borrower checks out the books.

SubFlows:

Alternate/Exceptional Flows:
- 4a. The ID Card is invalid, the book request is rejected.
- 5a. The Borrower either has overdue books fines, or both, the book request is rejected.

FIGURE 5-18 Flow Descriptions for the Borrow Books Use Case (Figure 4-21)



Use Case Name: Borrow Books	ID: 2	Importance Level: High
Primary Actor: Borrower	Use Case Type: Detail, Essential	

Stakeholders and Interests:
Borrower—wants to check out books
Librarian—wants to ensure borrower only gets books deserved

Brief Description: This use case describes how books are checked out of the library.

Trigger: Borrower brings books to check out desk.

Type: External

Relationships:
- Association: Borrower, Personnel Office, Registrar's Office
- Include:
- Extend:
- Generalization :

FIGURE 5-19 Overview Description for the Borrow Books Use Case (Figure 4-20)

also suggests that we should review the overview section of the use-case description (see Figure 5-19). In this case, the only additional information gleaned from the use-case description is the possible inclusion of classes for Personnel Office and Registrar's Office. This same process would also be completed for the remaining use cases contained in the functional model: Process Overdue Books, Maintain Book Collection, Search Collection, and Return Books (see Figure 4-6). Since we did not discuss these use cases in the previous chapter, we will review the problem description as a basis for beginning the next step (see Figure 5-20).

FIGURE 5-20
Overview Description of the Library Book Collection Management System

The functional requirements for an automated university library circulation system include the need to support searching, borrowing, and book-maintenance activities. The system should support searching by title, author, keywords, and ISBN. Searching the library's collection database should be available on terminals in the library and available to potential borrowers via the World Wide Web. If the book of interest is currently checked out, a valid borrower should be allowed to request the book to be returned. Once the book has been checked back in, the borrower requesting the book should be notified of the book's availability.

The borrowing activities are built around checking books out and returning books by borrowers. There are three types of borrowers: students, faculty and staff, and guests. Regardless of the type of borrower, the borrower must have a valid ID card. If the borrower is a student, having the system check with the registrar's student database validates the ID card. If the borrower is a faculty or staff member, having the system check with the personnel office's employee database validates the ID card. If the borrower is a guest, the ID card is checked against the library's own borrower database. If the ID card is valid, the system must also check to determine whether the borrower has any overdue books or unpaid fines. If the ID card is invalid, the borrower has overdue books, or the borrower has unpaid fines, the system must reject the borrower's request to check out a book; otherwise the borrower's request should be honored. If a book is checked out, the system must update the library's collection database to reflect the book's new status.

The book-maintenance activities deal with adding and removing books from the library's book collection. This requires a library manager to both logically and physically add and remove the book. Books being purchased by the library or books being returned in a damaged state typically cause these activities. If a book is determined to be damaged when it is returned and it needs to be removed from the collection, the last borrower will be assessed a fine. However, if the book can be repaired, depending on the cost of the repair, the borrower might not be assessed a fine. Finally, every Monday, the library sends reminder e-mails to borrowers who have overdue books. If a book is overdue more than two weeks, the borrower is assessed a fine. Depending on how long the book remains overdue, the borrower can be assessed additional fines every Monday.

The second step is to review the CRC cards to determine if there is any information missing. In the case of the library system, because we only used the Borrow Books use-case description, some information is obviously missing. By reviewing Figure 5-20, we see that we need to include the ability to search the book collection by title, author, keywords, and ISBN. This obviously implies a Book Collection class with four different search operations: Search By Title, Search By Author, Search By Keywords, and Search By ISBN. Interestingly, the description also implies either a set of subclasses or states for the Book class: Checked Out, Overdue, Requested, Available, and Damaged. We will return to the issue of states versus subclasses in the next chapter. The description implies many additional operations, including Returning Books, Requesting Books, Adding Books, Removing Books, Repairing Books, Fining Borrowers, and Emailing Reminders.

Next, we should use our own library experience to brainstorm potential additional classes, attributes, operations, and relationships that could be useful to include in the Library Book Collection Management System. In our library, there is also the need to Retrieve Books From Storage, Move Books to Storage, Request Books from the Interlibrary Loan System, Return Books to the Interlibrary Loan System, and Deal with E-Books. You also could include classes for Journals, DVDs, and other media. As you can see, many classes, attributes, operations, and relationships can be identified.

The third step, role-playing the CRC cards, requires us to apply the three role-playing steps described earlier:

- Review Use Cases
- Identify Relevant Actors and Objects
- Role Play Scenarios

For our purposes, we will use the Borrow Books use case to demonstrate. The relevant actors include Student Borrowers, Faculty/Staff Borrowers, Guest Borrowers, Librarians, Personnel Office, and Registrar's Office. These can be easily gleaned from the overview section of the use-case description (see Figure 5-19) and the use-case diagram (see Figure 4-6). The relevant objects seem to include Books, Borrower, and ID Card. Finally, to role-play the scenarios, we need to assign the roles to the different members of the team and try to perform each of the paths through the events of the use-case (see Figure 5-18). Based on the Events of the use case and the use case's activity diagram (see Figure 5-21), we can quickly identify nine scenarios, three for each type of Borrower (Student, Faculty/Staff, and Guest): Valid ID and No Overdue Books & No Fines, Valid ID only, and no Valid ID. When role-playing these scenarios, one question arises: What happens to the books that are requested when the request is rejected? Based on the current functional and structural models, the books are left sitting on the check out desk. That doesn't quite seem right. In reality, the books are reshelved. In fact, the notion of reshelving books is also relevant to when books are checked back in or after books have been repaired. Furthermore, the idea of adding books to the collection should also include the operation of shelving the books. As you should readily see, building structural models will also help uncover behavior that was omitted when building the functional models. Remember, object-oriented systems development is not only use-case driven but also is incremental and iterative.

The fourth step is to put everything together and to draw the class diagram. Figure 5-22 represents the first cut at drawing the class diagram for the Library Book Collection Management System. The classes identified in the previous steps have been linked with other classes via association, aggregation, and generalization relationships. For simplicity purposes, we only show the classes and their relationships; not their attributes, operations, or even the multiplicities on the association relationships.

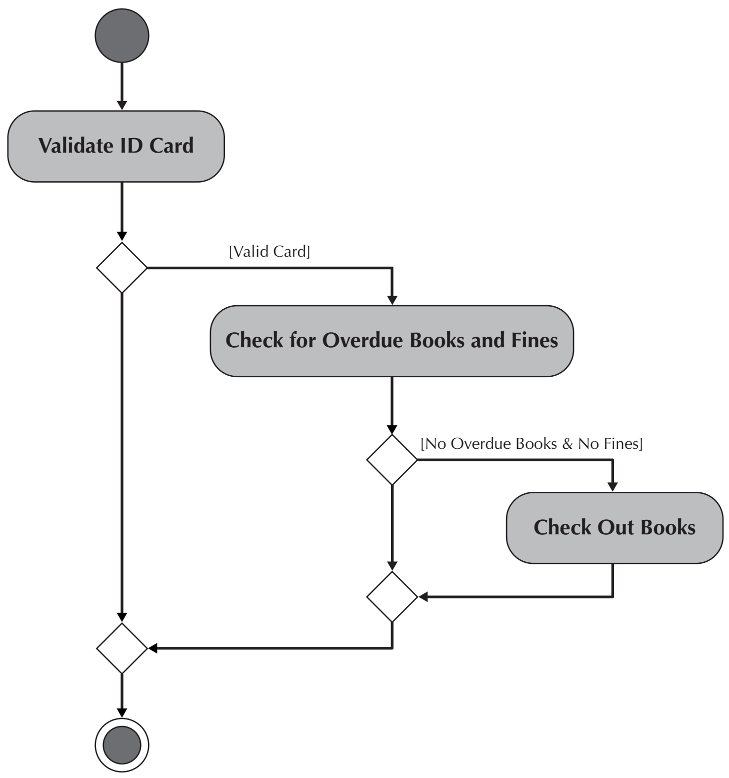

Validate ID Card
[Valid Card]
Check for Overdue Books and Fines
[No Overdue Books & No Fines]
Check Out Books

FIGURE 5-21
Activity Diagram for the Borrow Books Use Case (Figure 4-12)

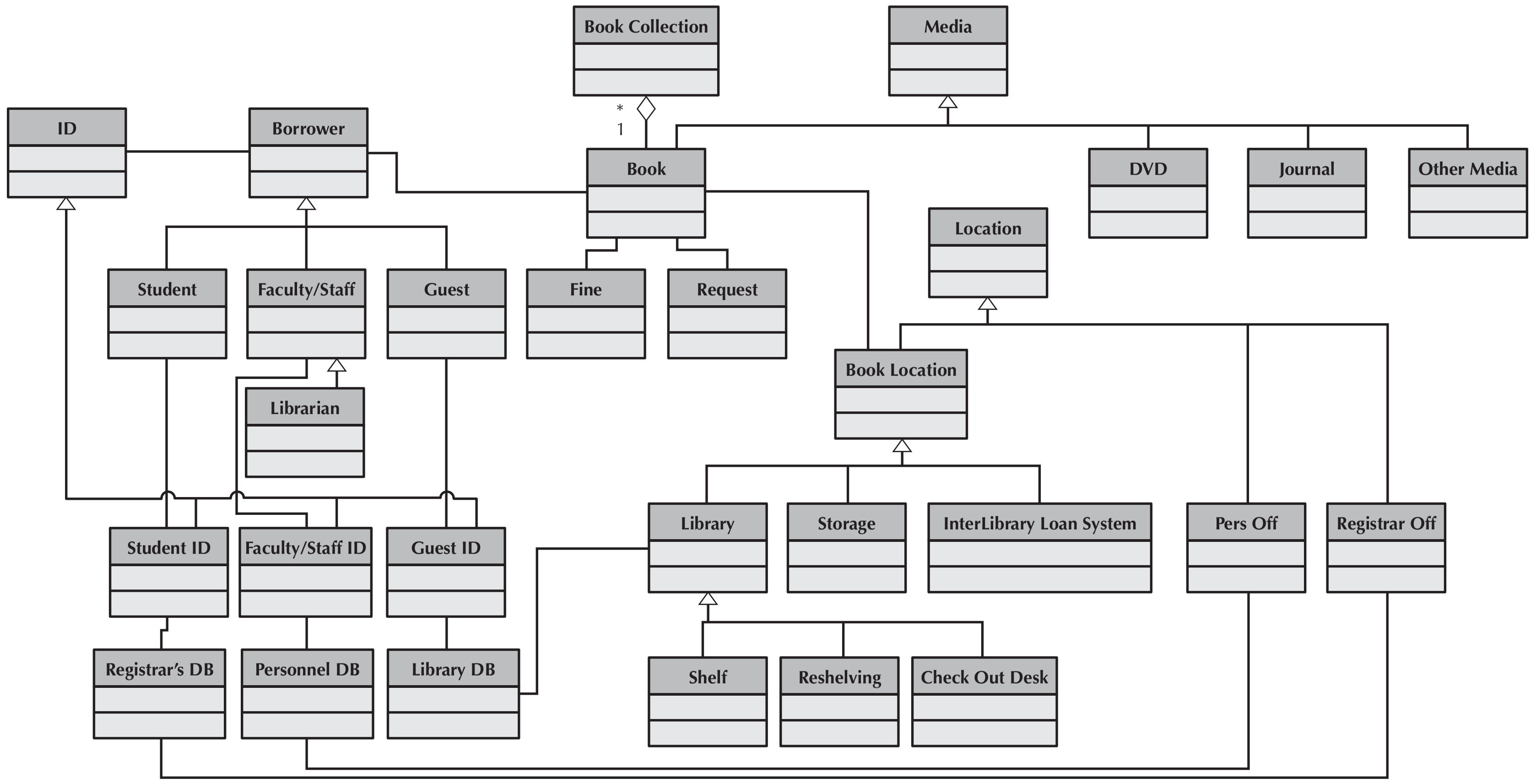


FIGURE 5-22 First-Cut Class Diagram for the Library Book Collection System

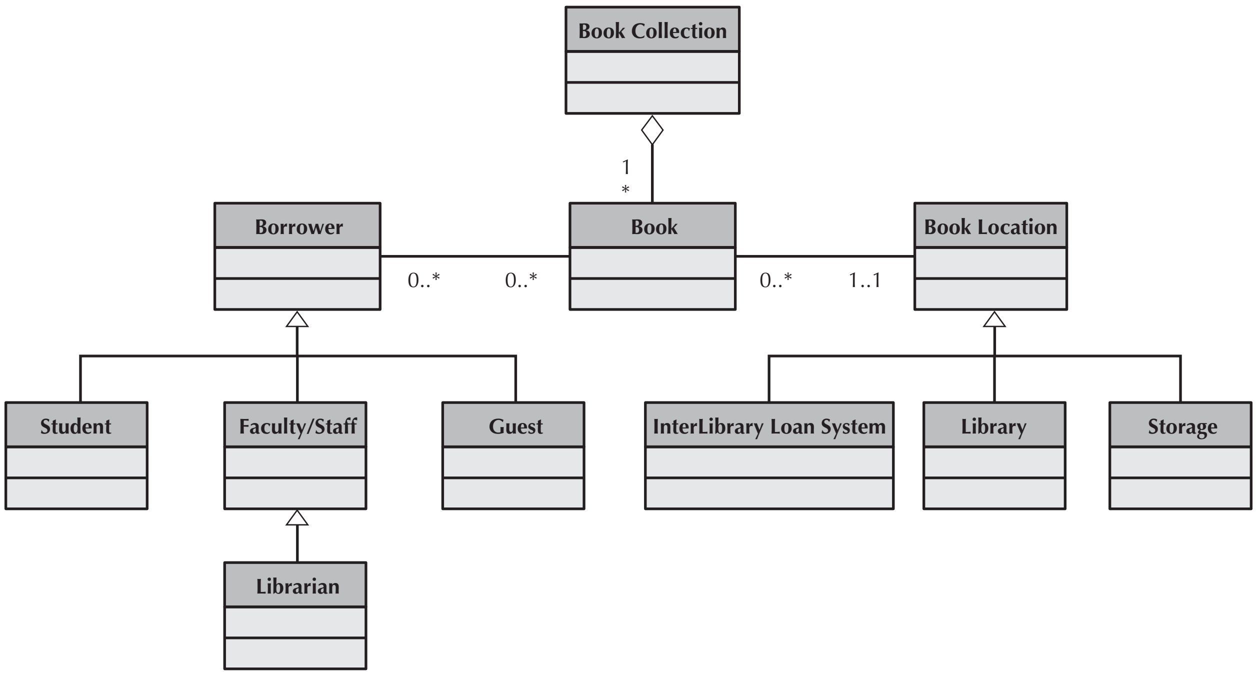


FIGURE 5-23 Second-Cut Class Diagram for the Library Book Collection System

The fifth step is to carefully review what has been created. Not only should you look for any missing classes, attributes, operations, and/or relationships, but you should also challenge every aspect of the current model. Specifically, are there classes, attributes, operations, and/or relationships that should be removed from the model? If so, there may be classes on the diagram that should have been modeled as attributes. For example, the Student, Fac/Staff, and Guest IDs should have been attributes with their respective classes. Furthermore, because this is a book collection management system, the inclusion of other media seems to be inappropriate. Finally, the Personnel Office and Registrar's Office are actually only actors in the system, not objects. Based on all of these deletions, a new version of the class diagram was drawn (see Figure 5-23). This diagram is much simpler and easier to understand.

The sixth step, incorporating useful patterns, enables us to take advantage of knowledge that was developed elsewhere. In this case, the pattern used in the library problem includes too many ideas that are not relevant to the current problem. However, by looking back to Figure 5-3, we see that one of the original patterns (the Place, Transaction, Participant, Transaction Line Item, and Item pattern—see the top left of the figure) is relevant. We incorporate that pattern into the class diagram by replacing Place by Check Out Desk, Participant by Borrower, Transaction by Check Out Trans, and Item by Book (Figure 5-24). Technically speaking, each of these replacements is simply a pattern customized to the problem at hand. We also then add the Transaction Line Item class that we had missed in the original structural model.

The seventh step is to review the current state of the structural model. Needless to say, the CRC card version and the class diagram version are no longer in agreement with each other. We return to this step in the next section of the chapter.

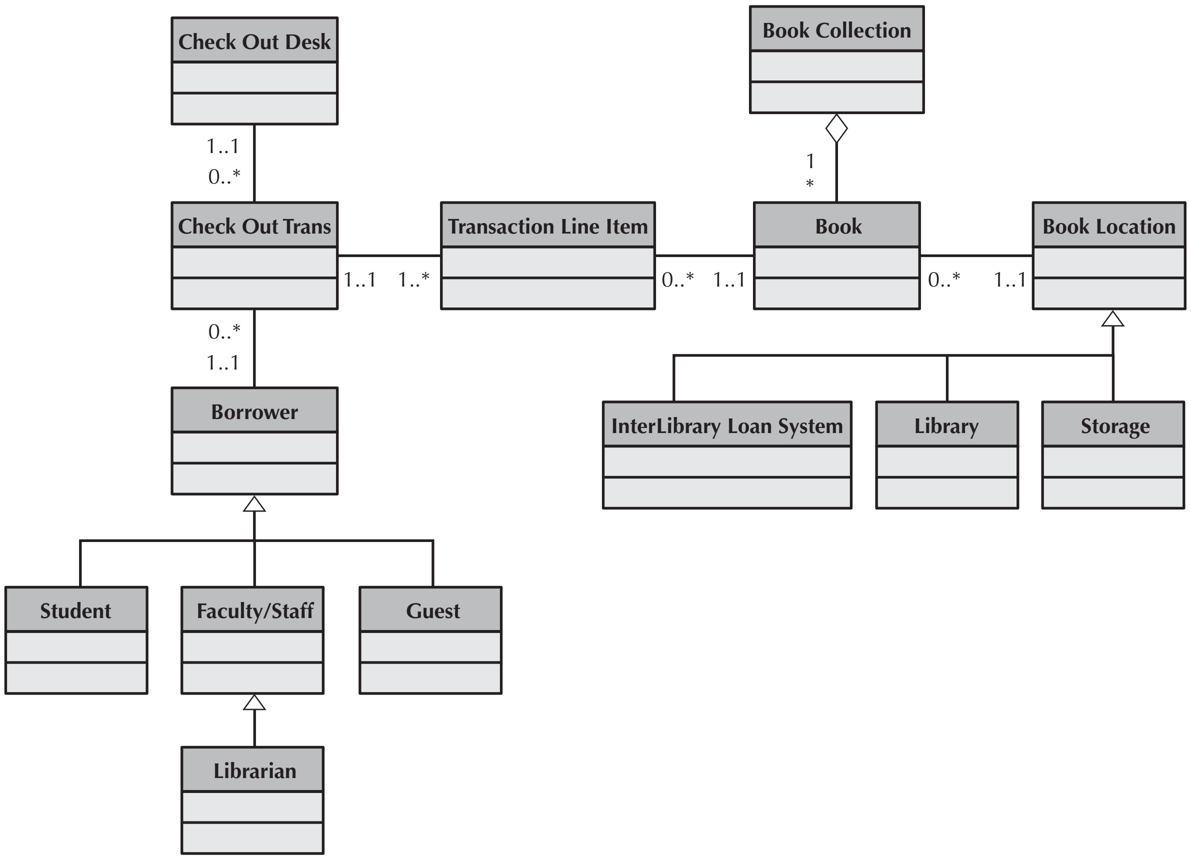


FIGURE 5-24 Class Diagram with Incorporated Pattern for the Library Book Collection System

VERIFYING AND VALIDATING THE STRUCTURAL MODEL[17]

Before we move on to creating behavioral models (see Chapter 6) of the problem domain, we need to verify and validate the structural model. In the previous chapter, we introduced the notion of walkthroughs as a way to verify and validate business processes and functional models. In this chapter, we combine walkthroughs with the power of role-playing as a way to more completely verify and validate the structural model that will underlie the business processes and functional models. In fact, all of the object identification approaches described in this chapter can be viewed as a way to test the fidelity of the structural model. Because we have already introduced the idea of role-playing the CRC cards and object identification, in this section we focus on performing walkthroughs.

In this case, the verification and validation of the structural model are accomplished during a formal review meeting using a walkthrough approach in which an analyst presents the model to a team of developers and users. The analyst walks through the model, explaining each part of the model and all the reasoning behind the decision to include each of the classes in the structural model. This explanation includes justifications for the attributes, operations, and relationships associated with the classes. Each class should be linked back to at least one use case; otherwise, the purpose of including the class in the structural model will not be

[17] The material in this section has been adapted from E. Yourdon, *Modern Structured Analysis* (Englewood Cliffs, NJ: Prentice Hall, 1989).

understood. Also including people outside the development team who produced the model can bring a fresh perspective to the model and uncover missing objects.

Previously, we suggested three representations that could be used for structural modeling: CRC cards, class diagrams, and object diagrams. Because an object diagram is simply an instantiation of some part of a class diagram, we limit our discussion to CRC cards and class diagrams. Similar to how we verified and validated the business process and functional models in the last chapter, we provide a set of rules that will test the consistency within the structural models. For example purposes, we use the appointment problem described in Chapter 4 and in this chapter. An example of the CRC card for the old patient class is shown in Figure 5-6, and the associated class diagram is portrayed in Figure 5-7.

First, every CRC card should be associated with a class on the class diagram, and vice versa. In the appointment example, the Old Patient class represented by the CRC card does not seem to be included on the class diagram. However, there is a Patient class on the class diagram (see Figures 5-6 and 5-7). The Old Patient CRC card most likely should be changed to simply Patient.

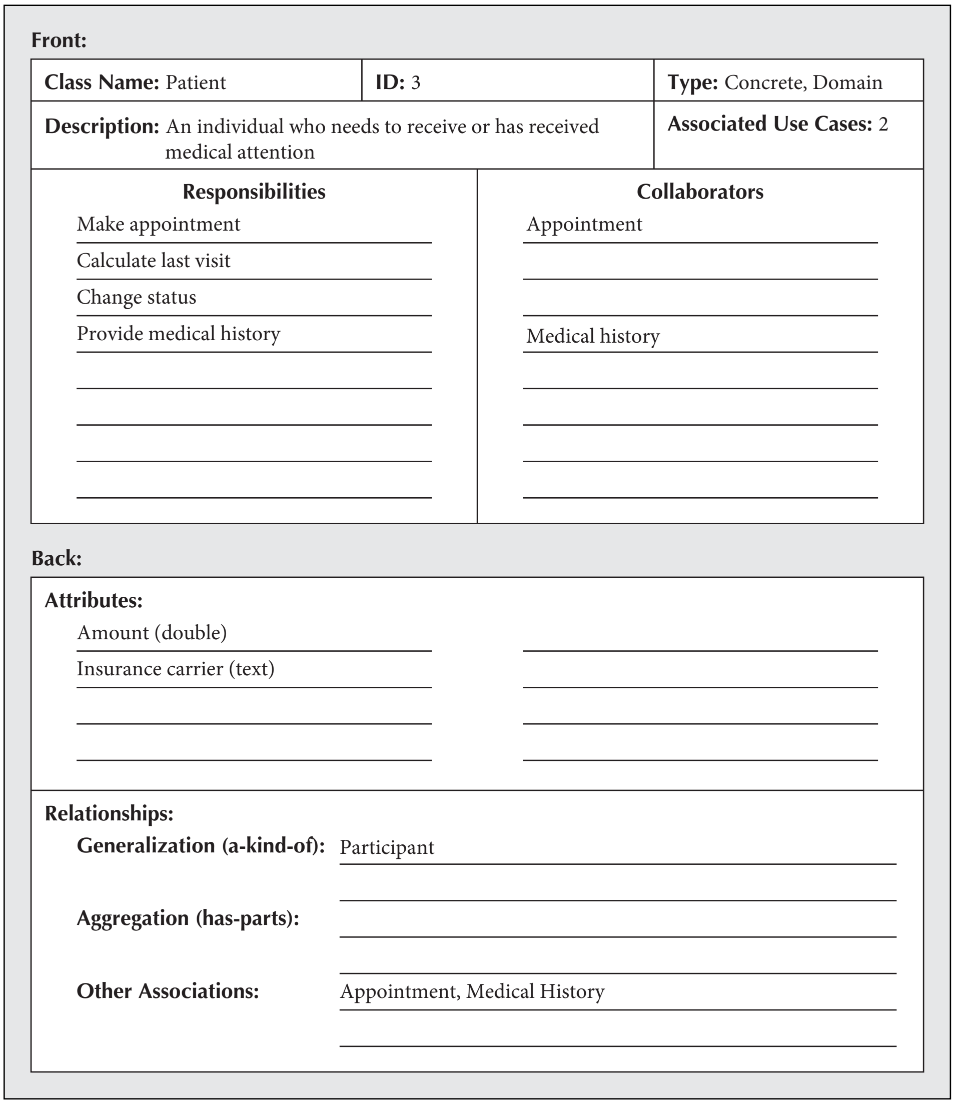
Second, the responsibilities listed on the front of the CRC card must be included as operations in a class on a class diagram, and vice versa. The make appointment responsibility on the new Patient CRC card also appears as the make appointment() operation in the Patient class on the class diagram. Every responsibility and operation must be checked.

Third, collaborators on the front of the CRC card imply some type of relationship on the back of the CRC card and some type of association that is connected to the associated class on the class diagram. The appointment collaborator on the front of the CRC card also appears as another association on the back of the CRC card and as an association on the class diagram that connects the Patient class with the Appointment class.

Fourth, attributes listed on the back of the CRC card must be included as attributes in a class on a class diagram, and vice versa. For example, the amount attribute on the new Patient CRC card is included in the attribute list of the Patient class on the class diagram.

Fifth, the object type of the attributes listed on the back of the CRC card and with the attributes in the attribute list of the class on a class diagram implies an association from the class to the class of the object type. For example, technically speaking, the amount attribute implies an association with the double type. However, simple types such as int and double are never shown on a class diagram. Furthermore, depending on the problem domain, object types such as Person, Address, or Date might not be explicitly shown either. However, if we know that messages are being sent to instances of those object types, we probably should include these implied associations as relationships.

Sixth, the relationships included on the back of the CRC card must be portrayed using the appropriate notation on the class diagram. For example in Figure 5-6, instances of the Patient class are *a-kind-of* Person, it has instances of the Medical History class as part of it, and it has an association with instances of the Appointment class. Thus, the association from the Patient class to the Person class should indicate that the Person class is a generalization of its subclasses, including the Patient class; the association from the Patient class to the Medical History class should be in the form of an aggregation association (a white diamond); and the association between instances of the Patient class and instances of the Appointment class should be a simple association. However, when we review the class diagram in Figure 5-7, this is not what we find. If you recall, we included in the class diagram the transaction pattern portrayed in Figure 5-4. When we did this, many changes were made to the classes contained in the class diagram. All of these changes should have been cascaded back through all of the CRC cards. In this case, the CRC card for the Patient class should show that a Patient is a-kind-of Participant (not Person) and that the relationship from Patient to Medical History should be a simple association (see Figure 5-25).



Front:

Class Name: Patient	**ID:** 3	**Type:** Concrete, Domain
Description: An individual who needs to receive or has received medical attention		**Associated Use Cases:** 2

Responsibilities	**Collaborators**
Make appointment	Appointment
Calculate last visit	
Change status	
Provide medical history	Medical history

Back:

Attributes:

Amount (double)

Insurance carrier (text)

Relationships:

Generalization (a-kind-of): Participant

Aggregation (has-parts):

Other Associations: Appointment, Medical History

FIGURE 5-25
Patient CRC Card

Seventh, an association class, such as the Treatment class in Figure 5-7, should be created only if there is indeed some unique characteristic (attribute, operation, or relationship) about the intersection of the connecting classes. If no unique characteristic exists, then the association class should be removed and only an association between the two connecting classes should be displayed.

Finally, as in the functional models, specific representation rules must be enforced. For example, a class cannot be a subclass of itself. The Patient CRC card cannot list Patient with the generalization relationships on the back of the CRC card, nor can a generalization relationship be drawn from the Patient class to itself. Again, all the detailed restrictions for each representation are beyond the scope of this book.[18] Figure 5-26 portrays the associations among the structural models.

[18] A good reference for these types of restrictions is S.W. Ambler, *The Elements of UML 2.0 Style* (Cambridge, UK: Cambridge University Press, 2005).

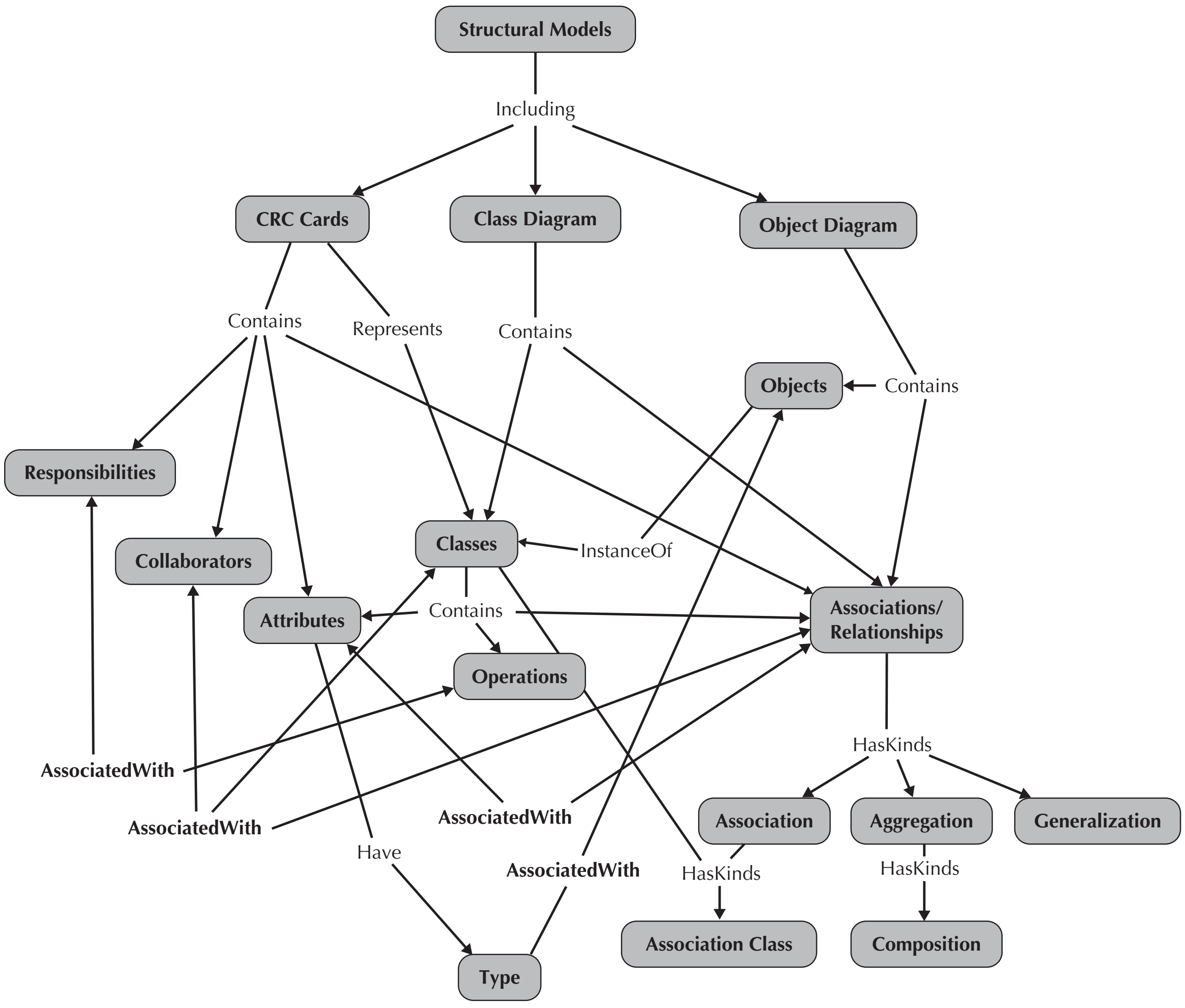


FIGURE 5-26 Interrelationships among Structural Models

APPLYING THE CONCEPTS AT PATTERSON SUPERSTORE

In Chapter 4, you learned how the functional models were developed in an iterative manner. After creating the functional model for the Mobile Scheduling (Version 1) of the Integrated Health Clinic Delivery System, the team had a good understanding of the business processes. Now it is time to identify the key data and to develop the structural model of the objects that support those business processes. Structural modeling for Mobile Scheduling (Version 1) involves creating, verifying, and validating CRC cards, class diagram, and object diagrams.

You can find the rest of the case at: www.wiley.com/go/dennis/casestudy

CHAPTER REVIEW

After reading and studying this chapter, you should be able to:

- ☐ Describe the purpose of a structural model.
- ☐ Describe the different elements of a structural model.
- ☐ Explain the difference between abstract and concrete classes.
- ☐ Describe the three general types of relationships typically used in a structural model.
- ☐ Create a structural model using textual analysis of use-case descriptions, brainstorming, common object lists, and patterns.
- ☐ Explain the purpose of a CRC card in structural modeling.
- ☐ Create a structural model using CRC cards.
- ☐ Describe the different elements of a CRC card.
- ☐ Describe how to role-play CRC cards using use-case scenarios.
- ☐ Describe the different elements of a class diagram.
- ☐ Describe the four basic operations that can be represented on a class diagram.
- ☐ Explain the differences between the types of relationships supported on a class diagram.
- ☐ Create a class diagram that represents a structural model.
- ☐ Describe the different elements of an object diagram.
- ☐ Create an object diagram that represents an instantiation of a portion of a class diagram.
- ☐ Verify and validate the evolving structural model using role-playing and walkthroughs.
- ☐ Verify and validate the functional model by ensuring the consistency of the three structural representations: CRC cards, class diagrams, and object diagrams.

KEY TERMS

A-kind-of
A-part-of
Abstract class
Aggregation association
Assemblies
Association
Association class
Attribute
Brainstorming
Class
Class diagram
Client
Collaboration
Common object list
Conceptual model
Concrete class
Constructor operation
Contract
Class–Responsibility–Collaboration (CRC)
CRC cards
Decomposition
Derived attribute
Doing responsibility
Destructor operation
Generalization association
Has-parts
Incidents
Instance
Interactions
Knowing responsibility
Method
Multiplicity
Object
Object diagram
Operation
Package
Parts
Pattern
Private
Protected
Public
Query operation
Responsibility
Role-playing
Roles
Server
State
Static model
Static structure diagram
Structural model
Subclass
Substitutability
Superclass
SVDPI
Tangible things
Textual analysis
Update operation
View
Visibility
Wholes

QUESTIONS

1. Describe to a businessperson the multiplicity of a relationship between two classes.
2. Why are assumptions important to a structural model?
3. What is an association class?
4. Contrast the following sets of terms: object, class, method, attribute, superclass, subclass, concrete class, abstract class.
5. Give three examples of derived attributes that may exist on a class diagram. How would they be denoted on the class diagram?

6. What are the different types of visibility? How would they be denoted on a class diagram?
7. Draw the relationships that are described by the following business rules. Include the multiplicities for each relationship.

   A patient must be assigned to only one doctor, and a doctor can have one or many patients.

   An employee has one phone extension, and a unique phone extension is assigned to an employee.

   A movie theater shows at least one movie, and a movie can be shown at up to four other movie theaters around town.

   A movie either has one star, two costars, or more than ten people starring together. A star must be in at least one movie.
8. How do you designate the reading direction of a relationship on a class diagram?
9. For what is an association class used in a class diagram? Give an example of an association class that may be found in a class diagram that captures students and the courses that they have taken.
10. Give two examples of aggregation, generalization, and association relationships. How is each type of association depicted on a class diagram?
11. Identify the following operations as constructor, query, or update. Which operations would not need to be shown in the class rectangle?

    Calculate employee raise (raise percent)
    Calculate sick days ()
    Increment number of employee vacation days ()
    Locate employee name ()
    Place request for vacation (vacation day)
    Find employee address ()
    Insert employee ()
    Change employee address ()
    Insert spouse ()
12. How are the different structural models related, and how does this affect verification and validation of the model?

EXERCISES

A. Create a CRC card for each of the following classes:

Movie (title, producer, length, director, genre)
Ticket (price, adult or child, showtime, movie)
Patron (name, adult or child, age)

B. Create a class diagram based on the CRC cards you created for exercise A.

C. Create a CRC card for each of the following classes. Consider that the entities represent a system for a patient billing system. Include only the attributes that would be appropriate for this context.

Patient (age, name, hobbies, blood type, occupation, insurance carrier, address, phone)
Insurance carrier (name, number of patients on plan, address, contact name, phone)
Doctor (specialty, provider identification number, golf handicap, age, phone, name)

D. Create a class diagram based on the CRC cards you created for exercise C.

E. Draw a class diagram for each of the following situations:

1. Whenever new patients are seen for the first time, they complete a patient information form that asks their name, address, phone number, and insurance carrier, which are stored in the patient information file. Patients can be signed up with only one carrier, but they must be signed up to be seen by the doctor. Each time a patient visits the doctor, an insurance claim is sent to the carrier for payment. The claim must contain information about the visit, such as the date, purpose, and cost. It would be possible for a patient to submit two claims on the same day.
2. The state of Georgia is interested in designing a system that will track its researchers. Information of interest includes researcher name, title, position, researcher's university name, university location, university enrollment, and researcher's research interests. Researchers are associated with one institution, and each researcher has several research interests.
3. A department store has a wedding registry. This registry keeps information about the customer (usually the bride), the products that the store carries, and the products for which each customer registers. Customers typically register for a large number of products, and many customers register for the same products.
4. Jim Smith's dealership sells Fords, Hondas, and Toyotas. In order to get in touch with these manufacturers easily, the dealership keeps information about each of them. The dealership keeps information about the models of cars from each

manufacturer, including dealer price, model name, and series (e.g., Honda, Civic, LX). Additionally, the dealership also keeps all sales information, including buyer's name, address and phone number, car purchased, and amount paid.

F. Create object diagrams based on the class diagrams you drew for exercise F.

G. Examine the class diagrams that you created for exercise F. How would the models change (if at all) based on these new assumptions?

1. Two patients have the same first and last names.
2. Researchers can be associated with more than one institution.
3. The store would like to keep track of purchase items.
4. Many buyers have purchased multiple cars from Jim over time because he is such a good dealer.

H. Visit a website that allows customers to order a product over the Web (e.g., Amazon.com). Create a structural model (CRC cards and class diagram) that the site must need to support its business process. Include classes to show what they need information about. Be sure to include the attributes and operations to represent the type of information they use and create. Finally, draw relationships, making assumptions about how the classes are related.

I. Using the seven-step process described in this chapter, create a structural model (CRC cards and class diagram) for exercise C in Chapter 4.

J. Perform a verification and validation walkthrough for the structural model created for exercise J.

K. Using the seven-step process described in this chapter, create a structural model for exercise E in Chapter 4.

L. Perform a verification and validation walkthrough for the structural model created for exercise L.

M. Using the seven-step process described in this chapter, create a structural model for exercise G in Chapter 4.

N. Perform a verification and validation walkthrough for the structural model created for exercise N.

O. Using the seven-step process described in this chapter, create a structural model for exercise I in Chapter 4.

P. Perform a verification and validation walkthrough for the structural model created for exercise P.

Q. Using the seven-step process described in this chapter, create a structural model for exercise L in Chapter 4.

R. Perform a verification and validation walkthrough for the structural model created for exercise R.

S. Using the seven-step process described in this chapter, create a structural model for exercise O in Chapter 4.

T. Perform a verification and validation walkthrough for the structural model created for exercise T.

U. Using the seven-step process described in this chapter, create a structural model for exercise R in Chapter 4.

V. Perform a verification and validation walkthrough for the structural model created for exercise V.

W. Using the seven-step process described in this chapter, create a structural model for exercise U in Chapter 4.

X. Perform a verification and validation walkthrough for the structural model created for exercise X.

MINICASES

1. West Star Marinas is a chain of twelve marinas that offer lakeside service to boaters; service and repair of boats, motors, and marine equipment; and sales of boats, motors, and other marine accessories. The systems development project team at West Star Marinas has been hard at work on a project that eventually will link all the marina's facilities into one unified, networked system.

   The project team has developed a use-case diagram of the current system. This model has been carefully checked. Last week, the team invited a number of system users to role-play the various use cases, and the use cases were refined to the users' satisfaction. Right now, the project manager feels confident that the as-is system has been adequately represented in the use-case diagram.

   The director of operations for West Star is the sponsor of this project. He sat in on the role-playing of the use cases and was very pleased by the thorough job the team had done in developing the model. He made it clear to you, the project manager, that he was anxious to see your team begin work on the use cases for the to-be system. He was a little skeptical that it was necessary for your team to spend any time modeling the current system in the first place but grudgingly admitted that the team really seemed to understand the business after going through that work.

The methodology you are following, however, specifies that the team should now turn its attention to developing the structural models for the as-is system. When you stated this to the project sponsor, he seemed confused and a little irritated. "You are going to spend even more time looking at the current system? I thought you were done with that! Why is this necessary? I want to see some progress on the way things will work in the future!"

What is your response to the director of operations? Why do we perform structural modeling? Is there any benefit to developing a structural model of the current system at all? How do the use cases and use-case diagram help us develop the structural model?

2. Holiday Travel Vehicles sells new recreational vehicles and travel trailers. When new vehicles arrive at Holiday Travel Vehicles, a new vehicle record is created. Included in the new vehicle record are a vehicle serial number, name, model, year, manufacturer, and base cost.

When a customer arrives at Holiday Travel Vehicles, he or she works with a salesperson to negotiate a vehicle purchase. When a purchase has been agreed upon, a sales invoice is completed by the salesperson. The invoice summarizes the purchase, including full customer information, information on the trade-in vehicle (if any), the trade-in allowance, and information on the purchased vehicle. If the customer requests dealer-installed options, they are listed on the invoice as well. The invoice also summarizes the final negotiated price, plus any applicable taxes and license fees. The transaction concludes with a customer signature on the sales invoice.

a. Identify the classes described in the preceding scenario (you should find six). Create CRC cards for each class.

Customers are assigned a customer ID when they make their first purchase from Holiday Travel Vehicles. Name, address, and phone number are recorded for the customer. The trade-in vehicle is described by a serial number, make, model, and year. Dealer-installed options are described by an option code, description, and price.

b. Develop a list of attributes for each class. Place the attributes onto the CRC cards.

Each invoice lists just one customer. A person does not become a customer until he or she purchases a vehicle. Over time, a customer may purchase a number of vehicles from Holiday Travel Vehicles.

Every invoice must be filled out by only one salesperson. A new salesperson might not have sold any vehicles, but experienced salespeople have probably sold many vehicles.

Each invoice only lists one new vehicle. If a new vehicle in inventory has not been sold, there will be no invoice for it. Once the vehicle sells, there will be just one invoice for it.

A customer may decide to have no options added to the vehicle or may choose to add many options. An option may be listed on no invoices, or it may be listed on many invoices.

A customer may trade in no more than one vehicle on a purchase of a new vehicle. The trade-in vehicle may be sold to another customer who later trades it in on another Holiday Travel vehicle.

c. Based on the preceding business rules in force at Holiday Travel Vehicles and CRC cards, draw a class diagram and document the relationships with the appropriate multiplicities. Remember to update the CRC cards.