CHAPTER 7

Moving on to Design

Object-oriented system development uses the requirements that were gathered during analysis to create a blueprint for the future system. A successful object-oriented design builds upon what was learned in earlier phases and leads to a smooth implementation by creating a clear, accurate plan of what needs to be done. This chapter describes the initial transition from analysis to design and presents three ways to approach the design for the new system.

OBJECTIVES

- Understand the verification and validation of the analysis models.
- Understand the transition from analysis to design.
- Understand the use of factoring, partitions, and layers.
- Be able to create package diagrams.
- Be familiar with the custom, packaged, and outsource design alternatives.
- Be able to create an alternative matrix.

INTRODUCTION

The purpose of analysis is to figure out what the business needs are. The purpose of design is to decide how to build the system. The major activity that takes place during *design* is evolving the set of analysis representations into design representations.

Throughout design, the project team carefully considers the new system with respect to the current environment and systems that exist within the organization as a whole. Major considerations in determining how the system will work include environmental factors, such as integrating with existing systems, converting data from legacy systems, and leveraging skills that exist in-house. Although the planning and analysis are undertaken to develop a possible system, the goal of design is to create a blueprint for a system that can be implemented.

An important initial part of design is to examine several design strategies and decide which will be used to build the system. Systems can be built from scratch, purchased and customized, or outsourced to others, and the project team needs to investigate the viability of each alternative. This decision influences the tasks that are to be accomplished during design.

At the same time, detailed design of the individual classes and methods that are used to map out the nuts and bolts of the system and how they are to be stored must still be completed. Techniques such as CRC cards, class diagrams, contract specification, method specification, and database design provide the final design details in preparation for the implementation

phase, and they ensure that programmers have sufficient information to build the right system efficiently. These topics are covered in Chapters 8 and 9.

Design also includes activities such as designing the user interface, system inputs, and system outputs, which involve the ways that the user interacts with the system. Chapter 10 describes these three activities in detail, along with techniques such as storyboarding and prototyping, which help the project team design a system that meets the needs of its users and is satisfying to use.

Finally, physical architecture decisions are made regarding the hardware and software that will be purchased to support the new system and the way that the processing of the system will be organized. For example, the system can be organized so that its processing is centralized at one location, distributed, or both centralized and distributed, and each solution offers unique benefits and challenges to the project team. Because global issues and security influence the implementation plans that are made, they need to be considered along with the system's technical architecture. Physical architecture, security, and global issues are described in Chapter 11.

The many steps of design are highly interrelated and, as with the steps in analysis, the analysts often go back and forth among them. For example, prototyping in the interface design step often uncovers additional information that is needed in the system. Alternatively, a system that is being designed for an organization that has centralized systems might require substantial hardware and software investments if the project team decides to change to a system in which all the processing is distributed.

PRACTICAL TIP

Avoiding Classic Design

In Chapter 2, we discussed several classic mistakes and how to avoid them. Here, we summarize four classic mistakes in design and discuss how to avoid them.

1. *Reducing design time:* If time is short, there is a temptation to reduce the time spent in "unproductive" activities such as design so that the team can jump into "productive" programming. This results in missing important details that have to be investigated later at a much higher time and cost (usually at least ten times higher).

   Solution: If time pressure is intense, use timeboxing to eliminate functionality or move it into future versions.

2. *Feature creep:* Even if you are successful at avoiding scope creep, about 25 percent of system requirements will still change. And, changes—big and small—can significantly increase time and cost.

   Solution: Ensure that all changes are vital and that the users are aware of the impact on cost and time. Try to move proposed changes into future versions.

3. *Silver bullet syndrome:* Analysts sometimes believe the marketing claims for some design tools that claim to solve all problems and magically reduce time and costs. No *one* tool or technique can eliminate overall time or costs by more than 25 percent (although some can reduce individual steps by this much).

   Solution: If a design tool has claims that appear too good to be true, just say no.

4. *Switching tools midproject:* Sometimes analysts switch to what appears to be a better tool during design in the hopes of saving time or costs. Usually, any benefits are outweighed by the need to learn the new tool. This also applies even to minor upgrades to current tools.

   Solution: Don't switch or upgrade unless there is a compelling need for *specific* features in the new tool, and then explicitly *increase* the schedule to include learning time.

Based upon material from Steve McConnell, *Rapid Development* (Redmond, WA: Microsoft Press, 1966).

VERIFYING AND VALIDATING THE ANALYSIS MODELS[1]

Before we evolve our analysis representations into design representations, we need to verify and validate the current set of analysis models to ensure that they faithfully represent the problem domain under consideration. This includes testing the fidelity of each model; for example, we must be sure that the activity diagram(s), use-case descriptions, and use-case diagrams all describe the same functional requirements. It also involves testing the fidelity between the models; for instance, transitions on a behavioral state machine are associated with operations contained in a class diagram. In Chapters 4, 5, and 6, we focused on verifying and validating the individual models: function, structural, and behavioral. In this chapter, we center our attention on ensuring that the different models are consistent. Figure 7-1 portrays the fact that the object-oriented analysis models are highly interrelated. For example, do the functional and structural models agree? What about the functional and behavioral models? And finally, are the structural and behavioral models trustworthy? In this section, we describe a set of rules that are useful to verify and validate the intersections of the analysis models. Depending on the specific constructs of each actual model, different interrelationships are relevant. The process of ensuring the consistency among them is known as *balancing the models*.

Balancing Functional and Structural Models

To balance the functional and structural models, we must ensure that the two sets of models are consistent with each other. That is, the activity diagrams, use-case descriptions, and use-case diagrams must agree with the CRC cards and class diagrams that represent the evolving model of the problem domain. Figure 7-2 shows the interrelationships between the functional and structural models. By reviewing this figure, we uncover four sets of associations between the models. This gives us a place to begin balancing the functional and structural models.[2]

First, every class on a class diagram and every CRC card must be associated with at least one use case, and vice versa. For example, the CRC card portrayed in Figure 7-3 and its related class contained in the class diagram (see Figure 7-4) are associated with the Make Old Patient Appt use case described in Figure 7-5.

Second, every activity or action contained in an activity diagram (see Figure 7-6) and every event contained in a use-case description (see Figure 7-5) should be related to one or more responsibilities on a CRC card and one or more operations in a class on a class diagram and vice versa. For example, the Get Patient Information activity on the example activity diagram (see Figure 7-6) and the first two events on the use-case description (see Figure 7-5) are associated with the make appointment responsibility on the CRC card (see Figure 7-3) and the makeAppointment() operation in the Patient class on the class diagram (see Figure 7-4).

Third, every object node on an activity diagram must be associated with an instance of a class on a class diagram (i.e., an object) and a CRC card or an attribute contained in a class and on a CRC card. However, in Figure 7-6, there is an object node, Appt Request Info, that does not seem to be related to any class in the class diagram portrayed in Figure 7-4. Thus, either the activity or class diagram is in error or the object node must represent an attribute. In this case, it does not seem to represent an attribute. We could add a class to the

[1] The material in this section is based upon material from E. Yourdon, *Modern Structured Analysis* (Englewood Cliffs, NJ: Prentice Hall, 1989). Verifying and validating are a type of testing. We also describe testing in Chapter 12.

[2] Role-playing the CRC cards (see Chapter 5) also can be very useful in verifying and validating the relationships among the functional and structural models.

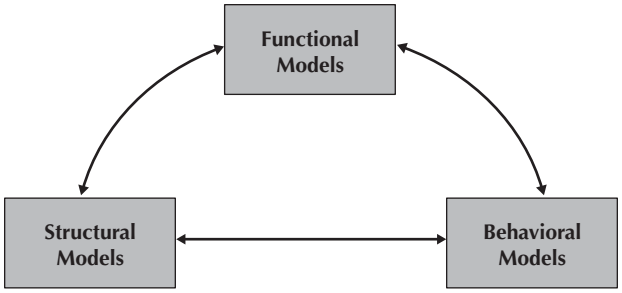


FIGURE 7-1
Object-Oriented Analysis Models

class diagram that creates temporary objects associated with the object node on the activity diagram. However, it is unclear what operations, if any, would be associated with these temporary objects. Therefore, a better solution would be to delete the Appt Request Info object nodes from the activity diagram. In reality, this object node represented only a set of bundled attribute values, i.e., data that would be used in the appointment system process (see Figure 7-7).

Fourth, every attribute and association/aggregation relationships contained on a CRC card (and connected to a class on a class diagram) should be related to the subject or object of an event in a use-case description. For example, in Figure 7-5, the second event states: The Patient provides the Receptionist with his or her name and address. By reviewing the CRC card in Figure 7-3 and the class diagram in Figure 7-4, we see that the Patient class is a subclass of the Participant class and hence inherits all the attributes, associations, and operations defined with the Participant class, where name and address attributes are defined.

Balancing Functional and Behavioral Models

As in balancing the functional and structural models, we must ensure the consistency of the two sets of models. In this case, the activity diagrams, use-case descriptions, and use-case diagrams must agree with the sequence diagrams, communication diagrams, behavioral state machines, and CRUDE matrix. Figure 7-8 portrays the relationships between the functional and behavioral models. Based on these interrelationships, we see that there are four areas with which we must be concerned.[3]

First, the sequence and communication diagrams must be associated with a use case on the use-case diagram and a use-case description. For example, the sequence diagram in Figure 7-9 and the communication diagram in Figure 7-10 are related to scenarios of the Make Old Patient Appt use case that appears in the use-case description in Figure 7-5 and the use-case diagram in Figure 7-11.

Second, actors on sequence diagrams, communication diagrams, and/or CRUDE matrices must be associated with actors on the use-case diagram or referenced in the use-case description, and vice versa. For example, the aPatient actor in the sequence diagram in Figure 7-9, the communication diagram in Figure 7-10, and the Patient row and column in the CRUDE matrix in Figure 7-12 appears in the use-case diagram in Figure 7-11 and the use-case description in Figure 7-5. However, the aReceptionist does not appear in the use-case diagram but is referenced in the events associated with the Make Old Patient Appt use-case description. In this case, the aReceptionist actor is obviously an internal actor, which cannot be portrayed on UML's use-case diagram.

[3] Performing CRUDE analysis (see Chapter 6) could also be useful in reviewing the intersections among the functional and behavioral models.

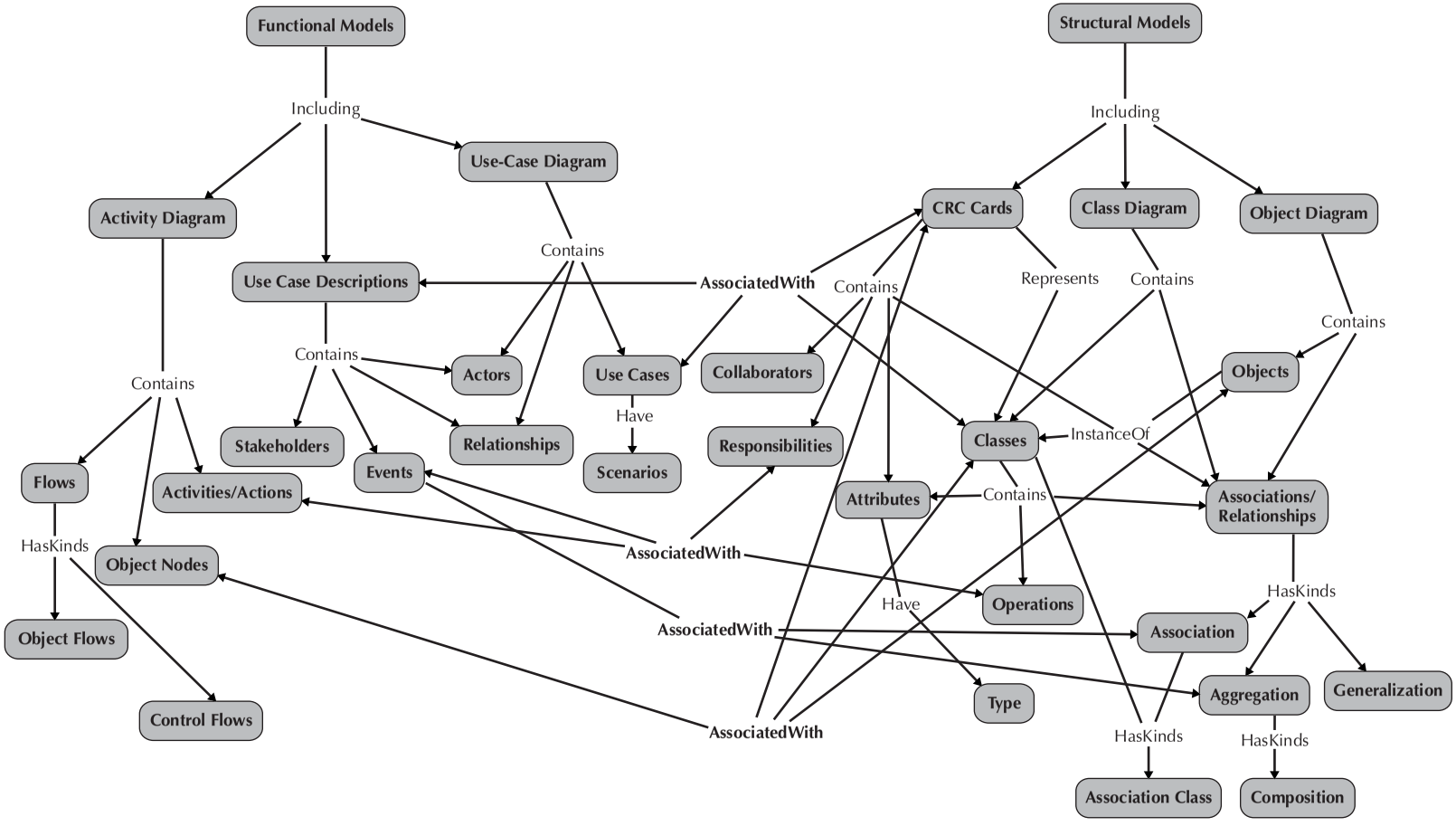


FIGURE 7-2 Relationships among Functional and Structural Models

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 7-3
Old Patient CRC Card (Figure 5-25)

Third, messages on sequence and communication diagrams, transitions on behavioral state machines, and entries in a CRUDE matrix must be related to activities and actions on an activity diagram and events listed in a use-case description, and vice versa. For example, the CreateAppt() message on the sequence and communication diagrams (see Figures 7-9 and 7-10) is related to the CreateAppointment activity (see Figure 7-7) and the S-1: New Appointment subflow on the use-case description (see Figure 7-5). The C entry in the Receptionist Appointment cell of the CRUDE matrix is also associated with these messages, activity, and subflow.

Fourth, all complex objects represented by an object node in an activity diagram must have a behavioral state machine that represents the object's lifecycle, and vice versa. As stated in Chapter 6, complex objects tend to be very dynamic and pass through a variety of states during their lifetimes. However, in this case because we no longer have any object nodes in the activity diagram (see Figure 7-7), there is no necessity for a behavioral state machine to be created based on the activity diagram.

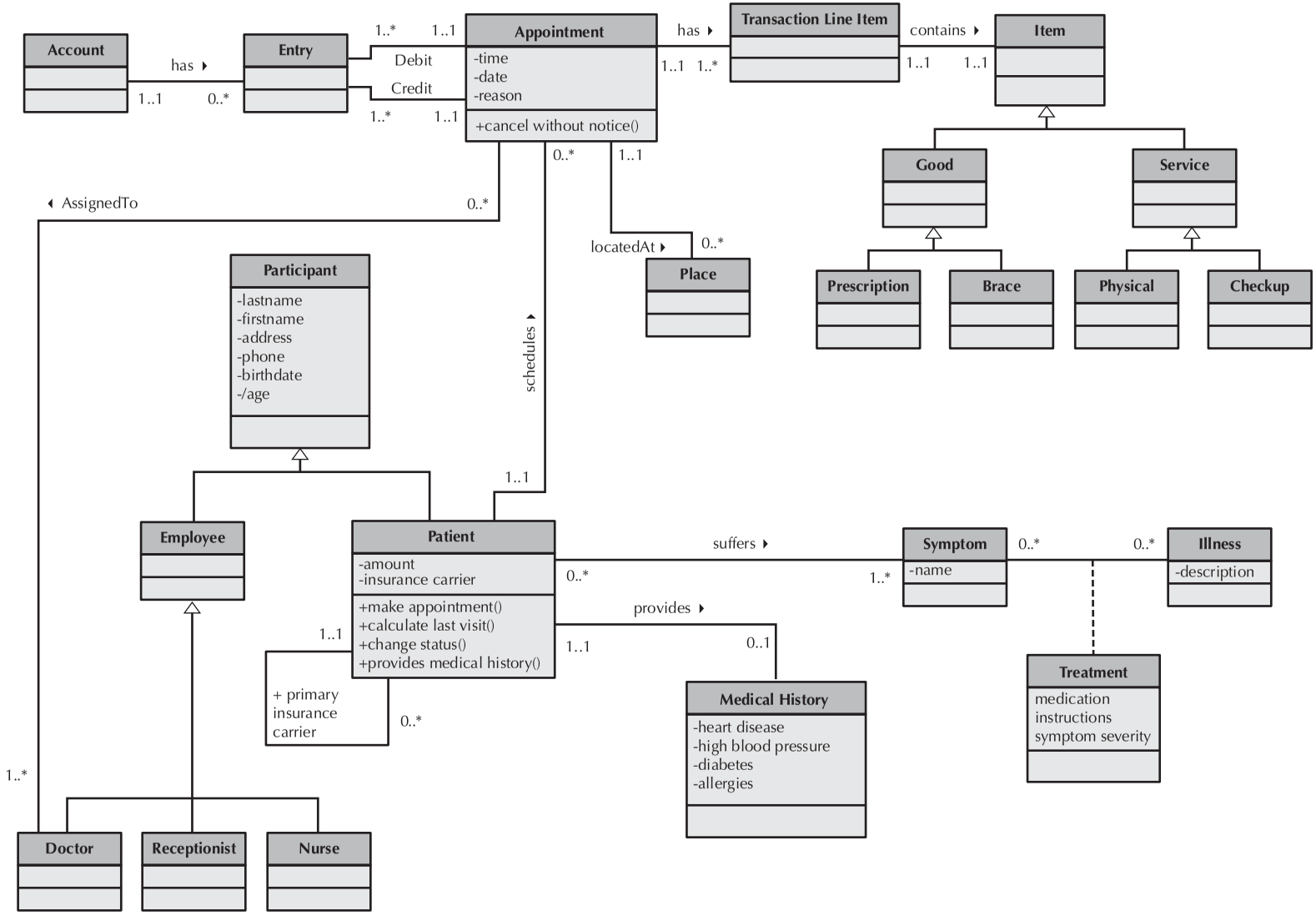


FIGURE 7-4 Appointment Problem Class Diagram (Figure 5-7)

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 7-5 Use-Case Description for the Make Old Patient Appt Use Case (Figure 4-13)

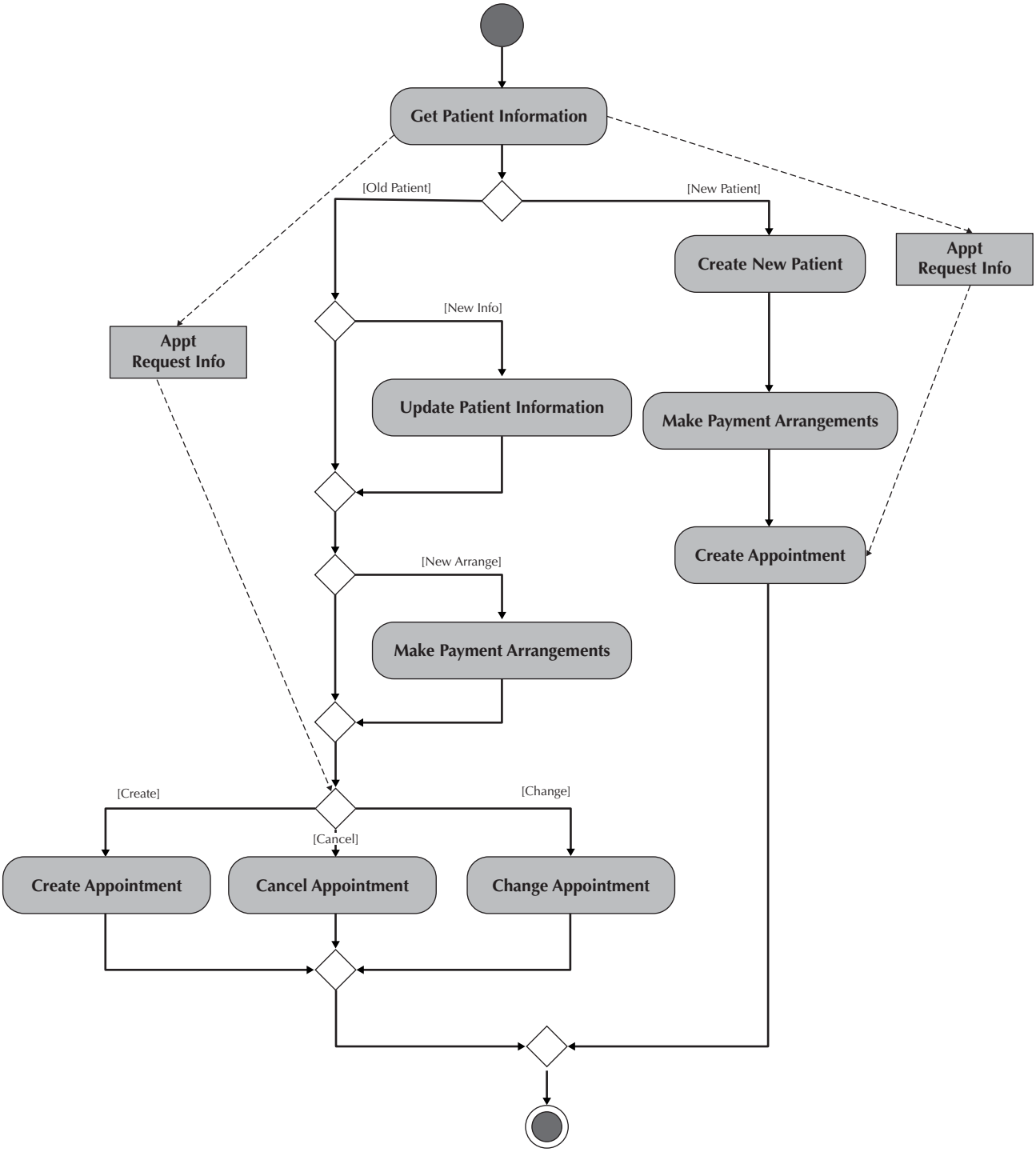


FIGURE 7-6 Activity Diagram for the Manage Appointments Use Case (Figure 4-8)

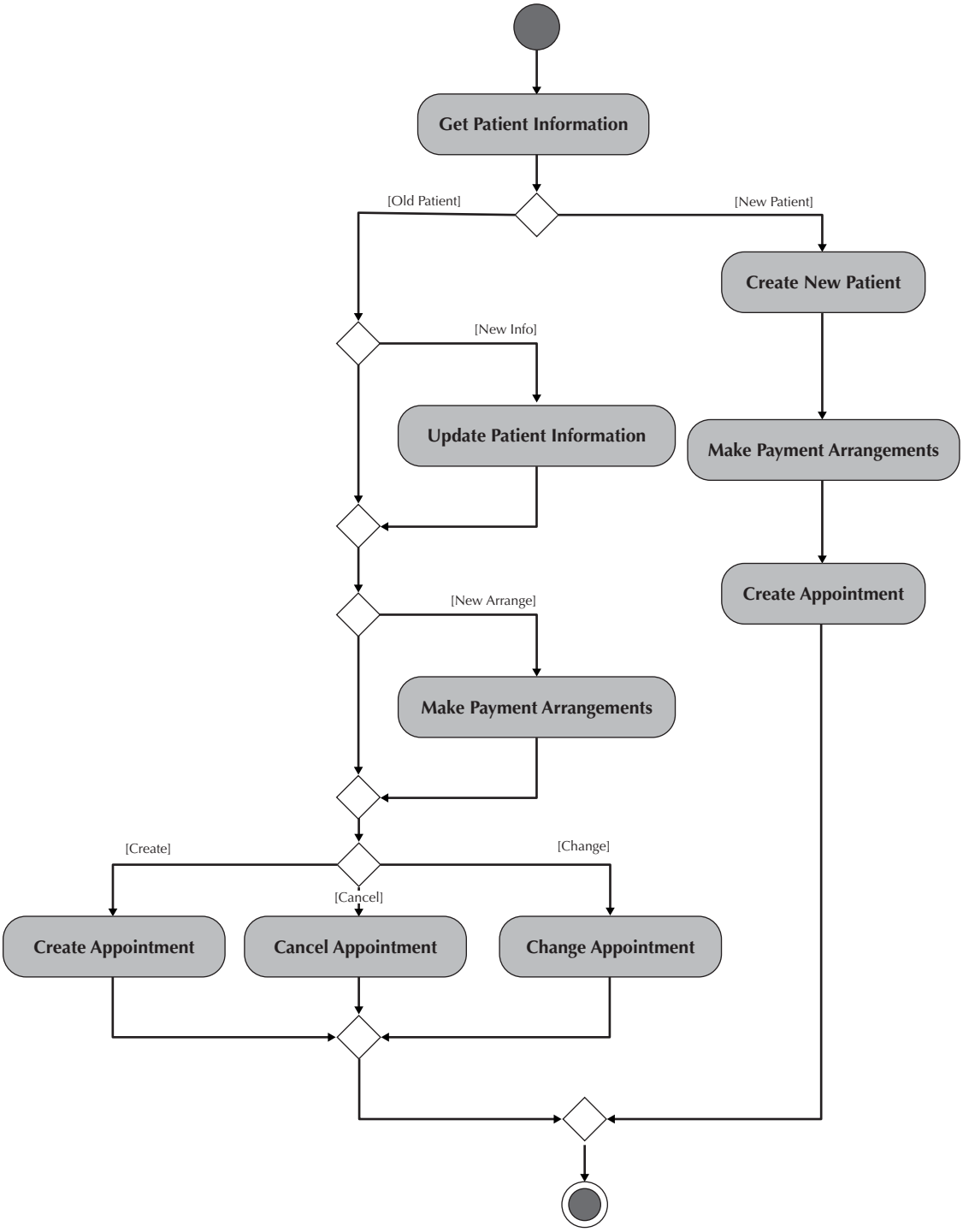


FIGURE 7-7 Corrected Activity Diagram for the Manage Appointments Use Case

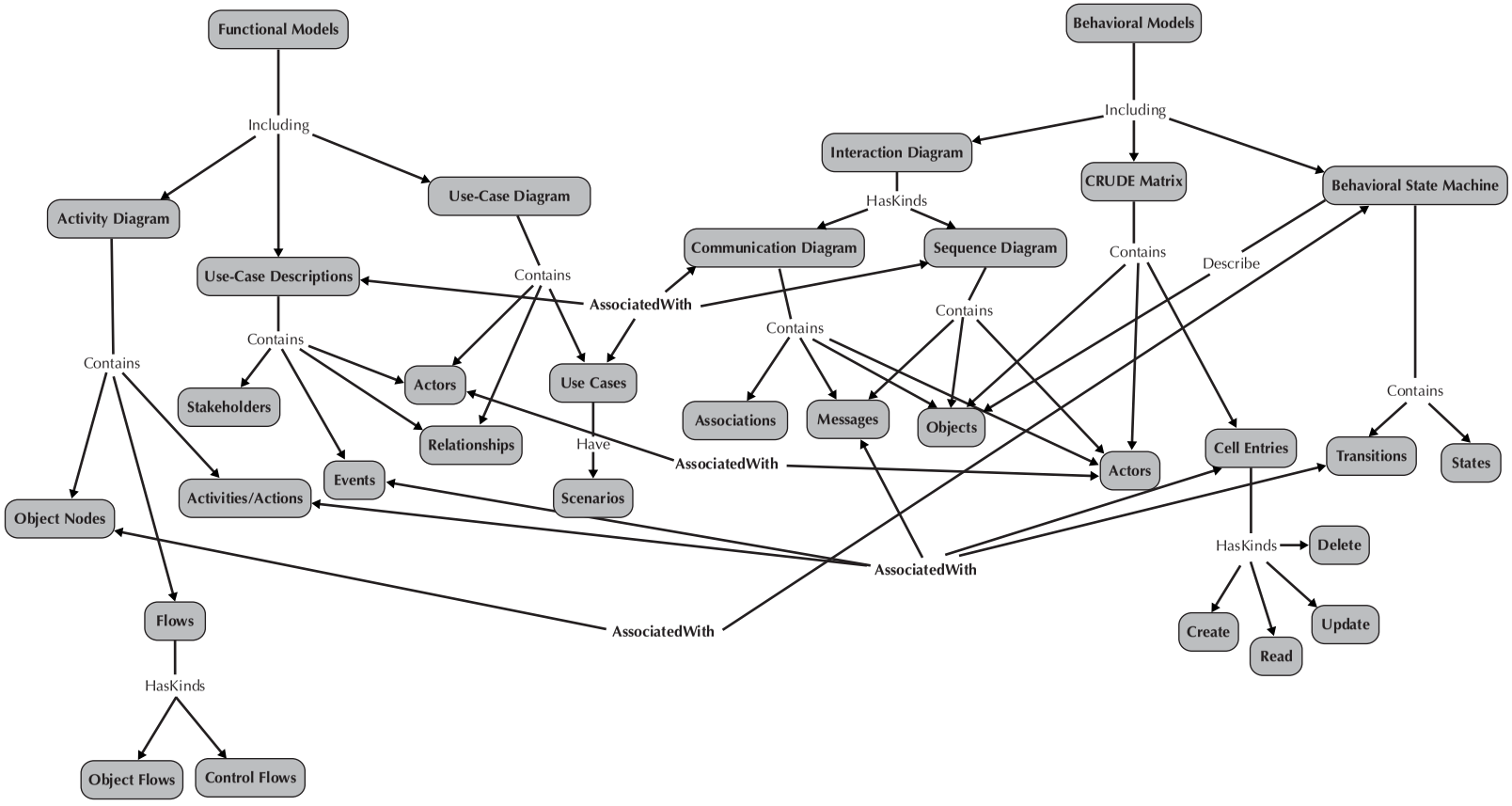


FIGURE 7-8 Relationships between Functional and Behavioral Models

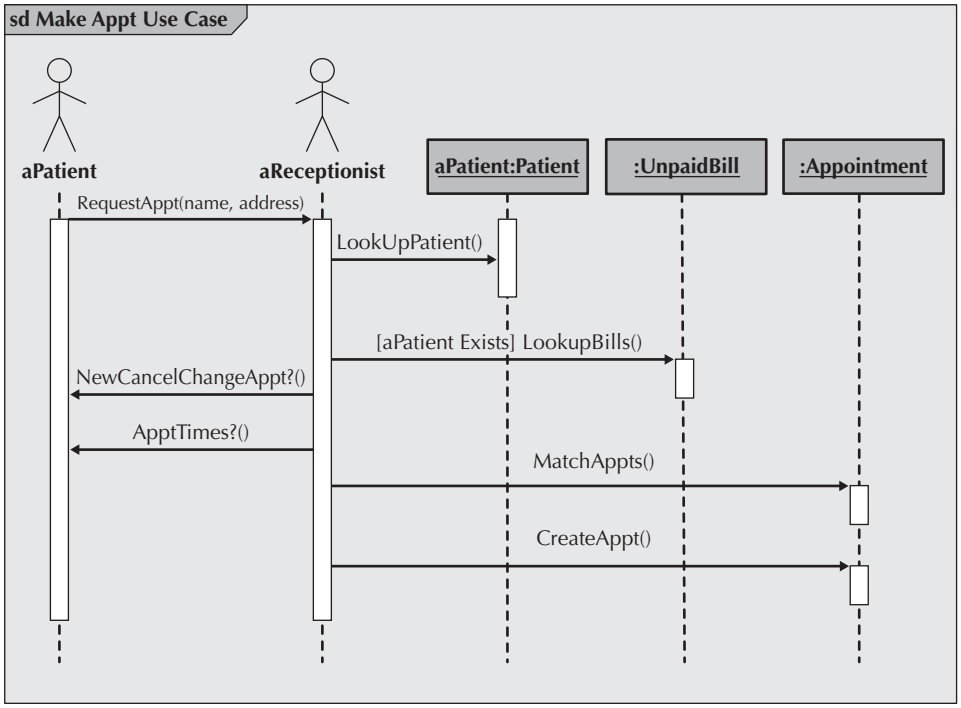


FIGURE 7-9
Sequence Diagram for a Scenario of the Make Old Patient Appt Use Case (Figure 6-1)

Balancing Structural and Behavioral Models

To discover the relationships between the structural and behavioral models, we use the concept map in Figure 7-13. In this case, there are five areas in which we must ensure the consistency between the models.[4]

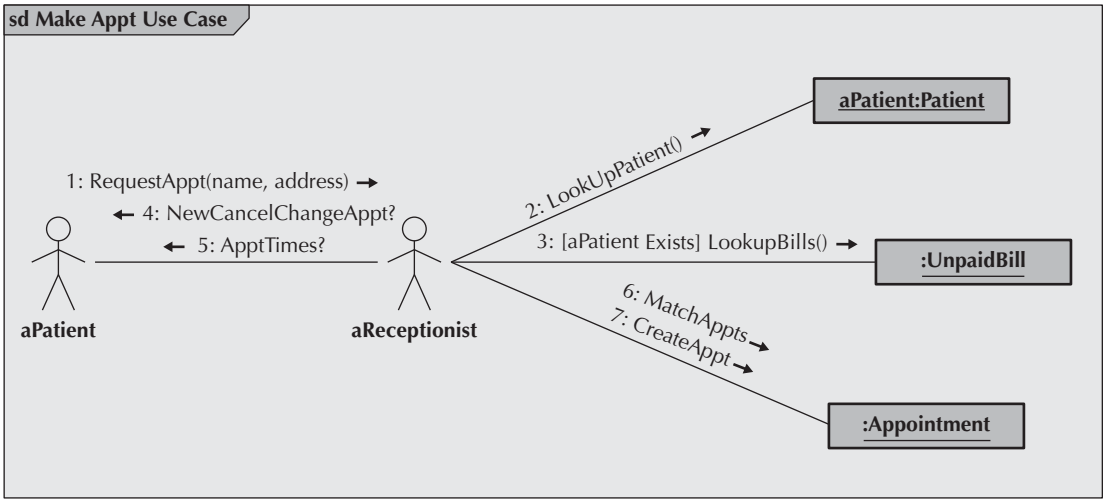


FIGURE 7-10 Communication Diagram for a Scenario of the Make Old Patient Appt Use Case (Figure 6-10)

[4] Role-playing (see Chapter 5) and CRUDE analysis (see Chapter 6) also can be very useful in this undertaking.

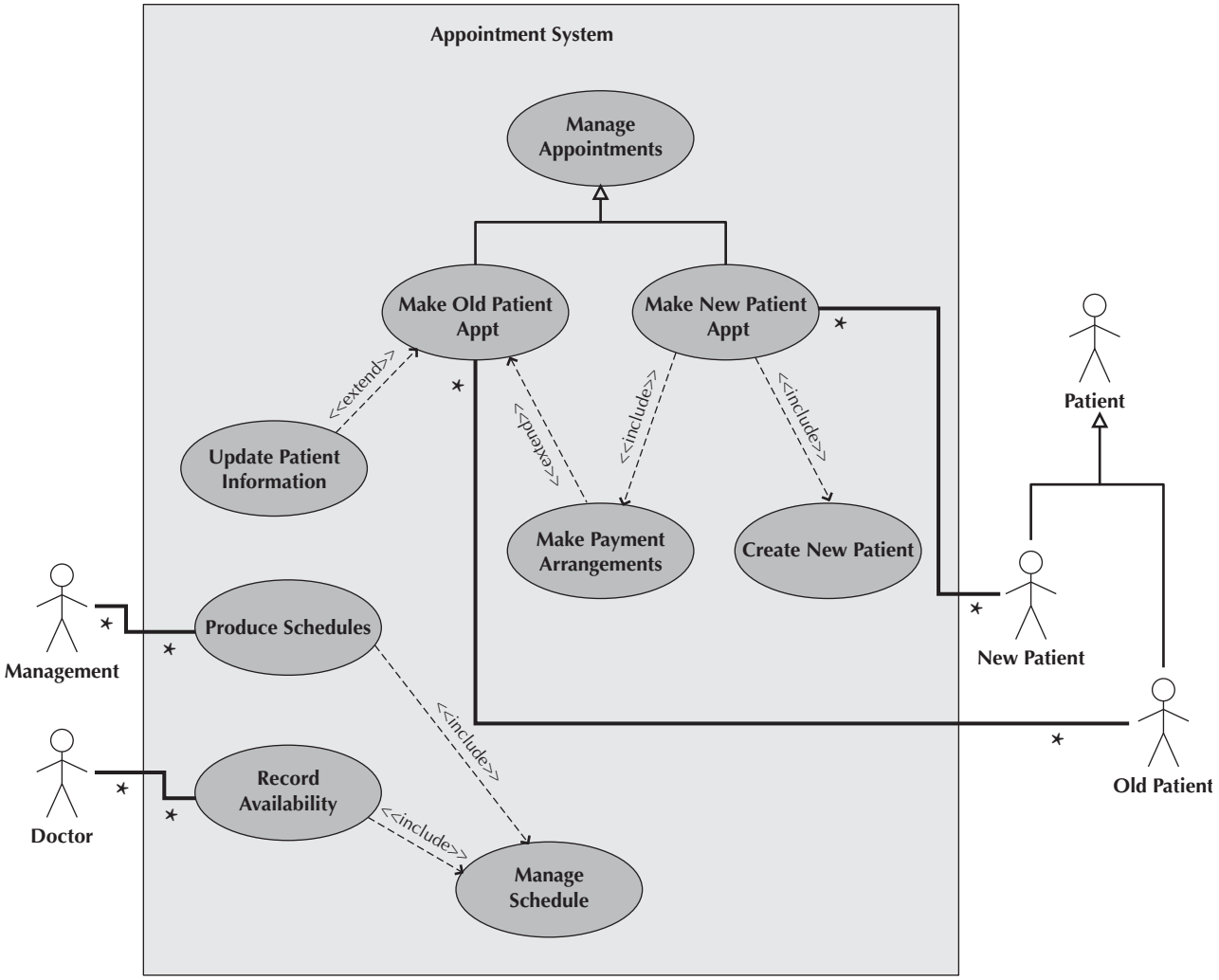


FIGURE 7-11 Modified Use-Case Diagram for the Appointment System (Figure 4-21)

	Receptionist	PatientList	Patient	UnpaidBills	Appointments	Appointment
Receptionist		RU	CRUD	R	RU	CRUD
PatientList			R			
Patient						
UnpaidBills						
Appointments						R
Appointment						

FIGURE 7-12 CRUDE Matrix for the Make Old Patient Apt Use Case (Figure 6-23)

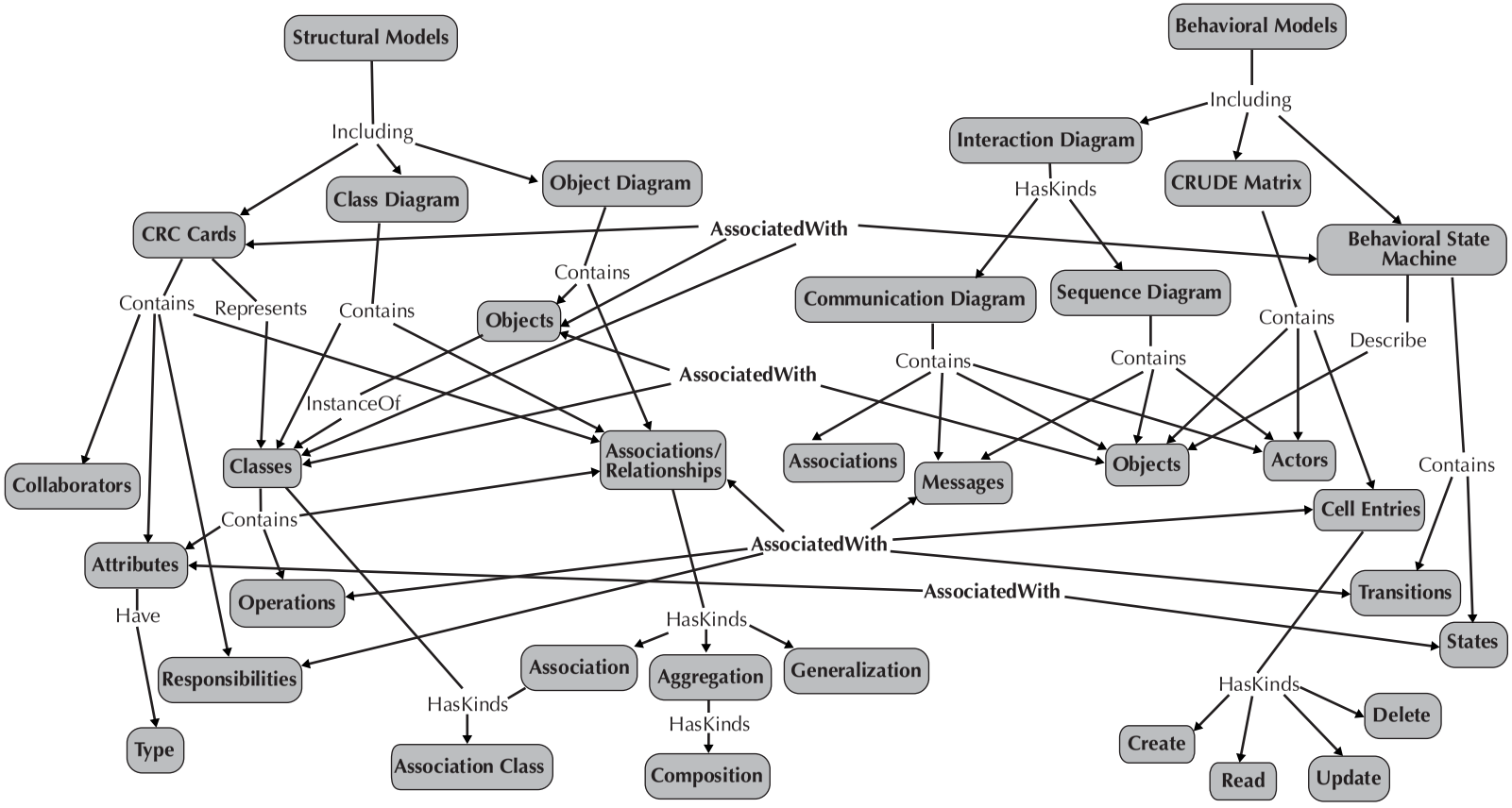


FIGURE 7-13 Relationships between Structural and Behavioral Models

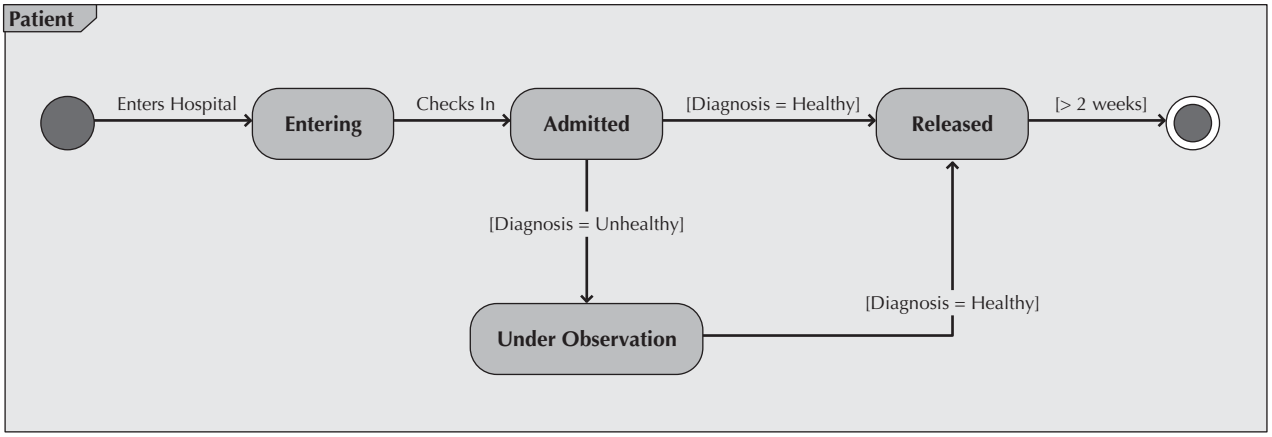


FIGURE 7-14 Behavioral State Machine for Hospital Patient (Figure 6-16)

First, objects that appear in a CRUDE matrix must be associated with classes that are represented by CRC cards and appear on the class diagram, and vice versa. For example, the Patient class in the CRUDE matrix in Figure 7-12 is associated with the CRC card in Figure 7-3 and the Patient class in the class diagram in Figure 7-4.

Second, because behavioral state machines represent the life cycle of complex objects, they must be associated with instances (objects) of classes on a class diagram and with a CRC card that represents the class of the instance. For example, the behavioral state machine that describes an instance of a Patient class in Figure 7-14 implies that a Patient class exists on a related class diagram (see Figure 7-4) and that a CRC card exists for the related class (see Figure 7-3).

Third, communication and sequence diagrams contain objects that must be an instantiation of a class that is represented by a CRC card and is located on a class diagram. For example, Figure 7-9 and Figure 7-10 have an anAppt object that is an instantiation of the Appointment class. Therefore, the Appointment class must exist in the class diagram (see Figure 7-4), and a CRC card should exist that describes it. However, there is an object on the communication and sequence diagrams associated with a class that did not exist on the class diagram: UnpaidBill. At this point, the analyst must decide to either modify the class diagram by adding these classes or rethink the communication and sequence diagrams. In this case, it is better to add the class to the class diagram (see Figure 7-15).

Fourth, messages contained on the sequence and communication diagrams, transitions on behavioral state machines, and cell entries on a CRUDE matrix must be associated with responsibilities and associations on CRC cards and operations in classes and associations connected to the classes on class diagrams. For example, the CreateAppt() message on the sequence and communication diagrams (see Figures 7-9 and 7-10) relate to the makeAppointment operation of the Patient class and the schedules association between the Patient and Appointment classes on the class diagram (see Figure 7-15).

Fifth, the states in a behavioral state machine must be associated with different values of an attribute or set of attributes that describe an object. For example, the behavioral state machine for the hospital patient object implies that there should be an attribute, possibly current status, which needs to be included in the definition of the class.

Summary

Figure 7-16 portrays a concept map that is a complete picture of the interrelationships among the diagrams covered in this section. It is obvious from the complexity of this

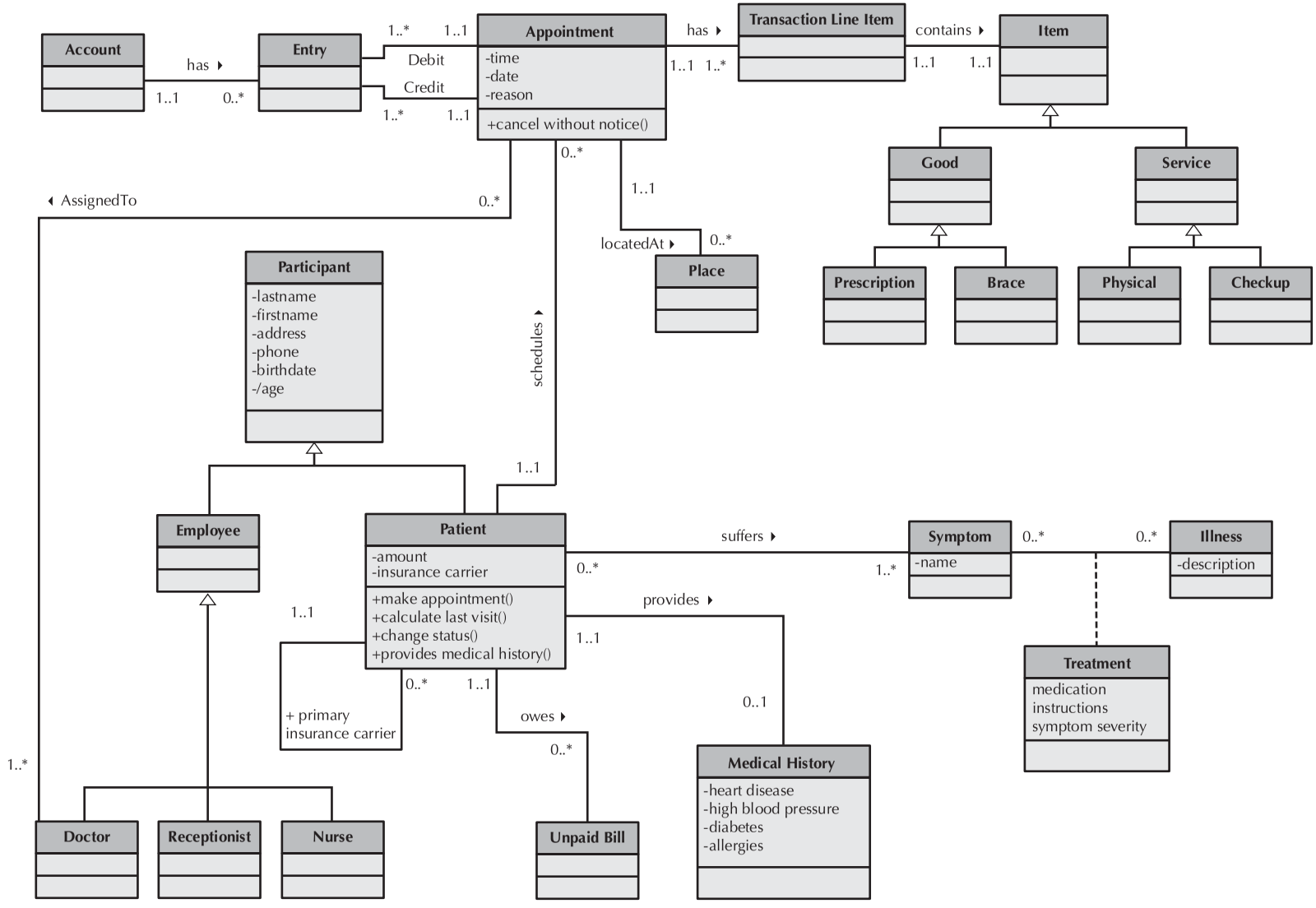


FIGURE 7-15 Corrected Appointment System Class Diagram

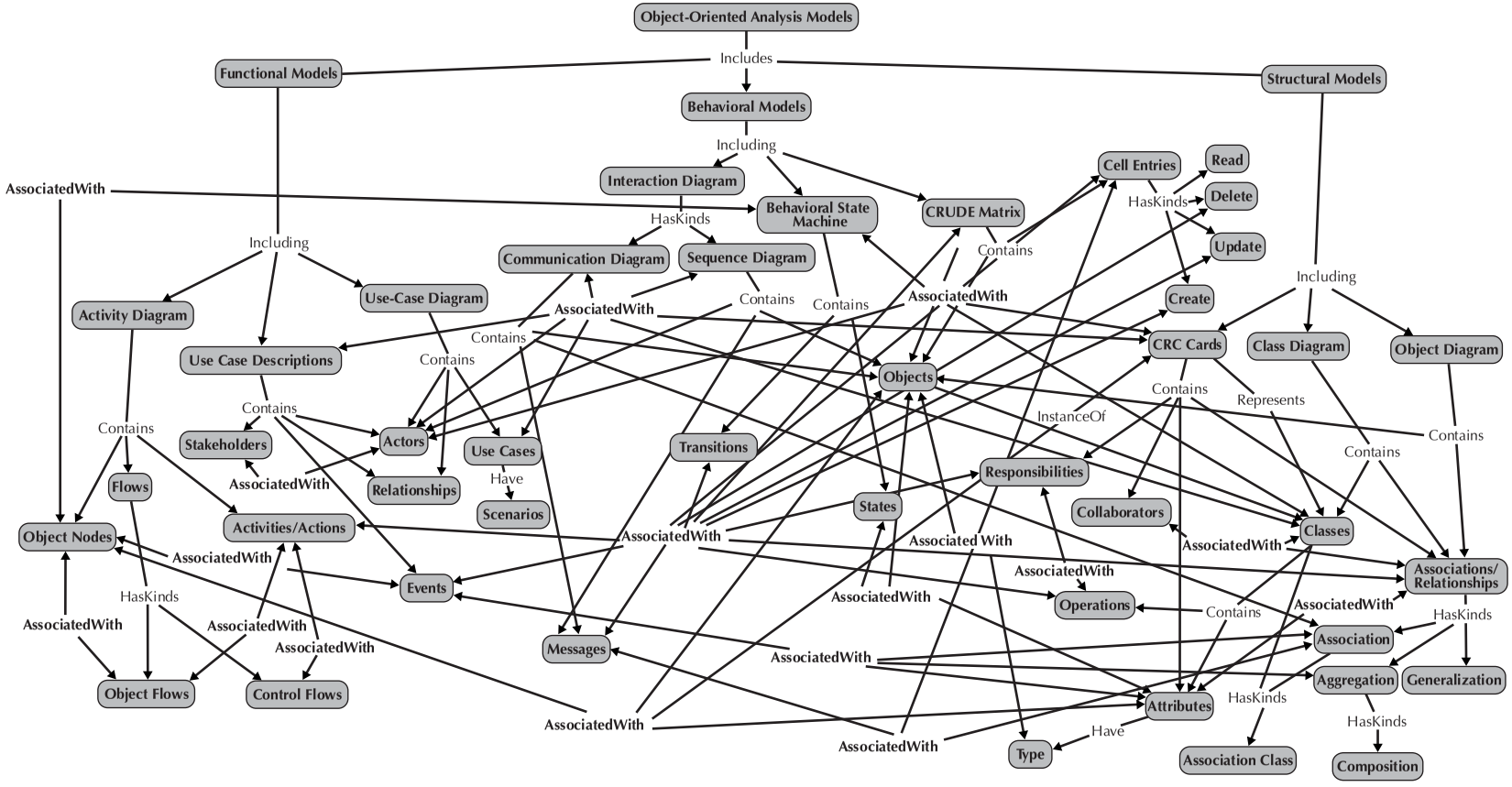


FIGURE 7-16 Interrelationships among Object-Oriented Analysis Models

figure that balancing all the functional, structural, and behavioral models is a very time-consuming, tedious, and difficult task. However, without paying this level of attention to the evolving models that represent the system, the models will not provide a sound foundation on which to design and build the system.

EVOLVING THE ANALYSIS MODELS INTO DESIGN MODELS

Now that we have successfully verified and validated our analysis models, we need to begin evolving them into appropriate design models. The purpose of the analysis models was to represent the underlying business problem domain as a set of collaborating objects. In other words, the analysis activities defined the functional requirements. To achieve this, the analysis activities ignored nonfunctional requirements such as performance and the system environment issues (e.g., distributed or centralized processing, user-interface issues, and database issues). In contrast, the primary purpose of the design models is to increase the likelihood of successfully delivering a system that implements the functional requirements in a manner that is affordable and easily maintainable. Therefore, in systems design, we address both the functional and nonfunctional requirements.

From an object-oriented perspective, system design models simply refine the system analysis models by adding system environment (or solution domain) details to them and refining the problem domain information already contained in the analysis models. When evolving the analysis model into the design model, you should first carefully review the use cases and the current set of classes (their operations and attributes and the relationships between them). Are all the classes necessary? Are there any missing classes? Are the classes fully defined? Are any attributes or methods missing? Do the classes have any unnecessary attributes and methods? Is the current representation of the evolving system optimal? Obviously, if we have already verified and validated the analysis models, quite a bit of this has already taken place. Yet, object-oriented systems development is both incremental and iterative. Therefore, we must review the analysis models again. However, this time we begin looking at the models of the problem domain through a design lens. In this step, we make modifications to the problem domain models that will enhance the efficiency and effectiveness of the evolving system.

In the following sections, we introduce factoring, partitions and collaborations, and layers as a way to evolve problem domain-oriented analysis models into optimal solution domain-oriented design models. From an enhanced Unified Process perspective (see Figure 1-16), we are moving from the analysis workflow to the design workflow, and we are moving further into the Elaboration phase and partially into the Construction phase.

Factoring

Factoring is the process of separating out a *module* into a stand-alone module. The new module can be a new *class* or a new *method.* For example, when reviewing a set of classes, it may be discovered that they have a similar set of attributes and methods. Thus, it might make sense to factor out the similarities into a separate class. Depending on whether the new class should be in a superclass relationship to the existing classes or not, the new class can be related to the existing classes through a *generalization (a-kind-of)* or possibly through an *aggregation (has-parts)* relationship. Using the appointment system example, if the Employee class had not been identified, we could possibly identify it at this stage by factoring out the similar methods and attributes from the Nurse, Receptionist, and Doctor classes. In this case, we would relate the new class (Employee) to the existing classes using the generalization (a-kind-of) relationship. Obviously, by extension we also could have created the Participant class if it had not been previously identified.

Abstraction and *refinement* are two processes closely related to factoring. Abstraction deals with the creation of a higher-level idea from a set of ideas. Identifying the Employee class is an example of abstracting from a set of lower classes to a higher one. In some cases, the abstraction process identifies *abstract classes,* whereas in other situations, it identifies additional *concrete classes.*[5] The refinement process is the opposite of the abstraction process. In the appointment system example, we could identify additional subclasses of the Employee class, such as Secretary and Bookkeeper. Of course we would add the new classes only if there were sufficient differences among them. Otherwise, the more general class, Employee, would suffice.

Partitions and Collaborations

Based on all the factoring, refining, and abstracting that can take place to the evolving system, the sheer size of the system representation can overload the user and the developer. At this point in the evolution of the system, it might make sense to split the representation into a set of *partitions.* A partition is the object-oriented equivalent of a subsystem,[6] where a subsystem is a decomposition of a larger system into its component systems (e.g., an accounting information system could be functionally decomposed into an accounts-payable system, an accounts-receivable system, a payroll system, etc.). From an object-oriented perspective, partitions are based on the pattern of activity (messages sent) among the objects in an object-oriented system. We describe an easy approach to model partitions and collaborations later in this chapter: packages and package diagrams.

A good place to look for potential partitions is the *collaborations* modeled in UML's communication diagrams (see Chapter 6). If you recall, one useful way to identify collaborations is to create a communication diagram for each use case. However, because an individual class can support multiple use cases, an individual class can participate in multiple use-case-based collaborations. In cases where classes are supporting multiple use cases, the collaborations should be merged. The class diagram should be reviewed to see how the different classes are related to one another. For example, if attributes of a class have complex object types, such as Person, Address, or Department, and these object types were not modeled as associations in the class diagram, we need to recognize these implied associations. Creating a diagram that combines the class diagram with the communication diagrams can be very useful to show to what degree the classes are coupled.[7] The greater the coupling between classes, the more likely the classes should be grouped together in a collaboration or partition. By looking at a CRUDE matrix, we can use CRUDE analysis (see Chapter 6) to identify potential classes on which to merge collaborations.

One of the easiest techniques to identify the classes that could be grouped to form a collaboration is through the use of cluster analysis or multiple dimensional scaling. These statistical techniques enable the team to objectively group classes together based on their affinity for each other. The affinity can be based on semantic relationships, different types of messages being sent between them (e.g., create, read, update, delete, or execute), or some weighted combination of both. There are many different similarity measures and many different algorithms on which the clusters can be based, so one must be careful when using these techniques. Always make sure that the collaborations identified using these techniques

[5] See Chapter 5 for the differences between abstract and concrete classes.

[6] Some authors refer to partitions as subsystems [e.g., see R. Wirfs-Brock, B. Wilkerson, and L. Weiner, *Designing Object-Oriented Software* (Englewood Cliffs, NJ: Prentice Hall, 1990)], whereas others refer to them as layers [e.g., see I. Graham, *Migrating to Object Technology* (Reading, MA: Addison-Wesley, 1994)]. However, we have chosen to use the term *partition* [C. Larman, *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design* (Englewood Cliffs, NJ: Prentice Hall, 1998)] to minimize confusion between subsystems in a traditional systems development approach and layers associated with Rational's Unified Approach.

[7] We describe the concept of coupling in Chapter 8.

make sense from the problem domain perspective. Just because a mathematical algorithm suggests that the classes belong together does not make it so. However, this is a good approach to create a first-cut set of collaborations.

Depending on the complexity of the merged collaboration, it may be useful in decomposing the collaboration into multiple partitions. In this case, in addition to having collaborations between objects, it is possible to have collaborations among partitions. The general rule is the more messages sent between objects, the more likely the objects belong in the same partition. The fewer messages sent, the less likely the two objects belong together.

Another useful approach to identifying potential partitions is to model each collaboration between objects in terms of clients, servers, and contracts. A *client* is an instance of a *class* that sends a *message* to an instance of another class for a *method* to be executed; a *server* is the instance of a class that receives the message; and a *contract* is the specification that formalizes the interactions between the client and server objects (see Chapters 5 and 8). This approach allows the developer to build up potential partitions by looking at the contracts that have been specified between objects. In this case, the more contracts there are between objects, the more likely that the objects belong in the same partition. The fewer contracts, the less chance there is that the two classes belong in the same partition.

Remember, the primary purpose of identifying collaborations and partitions is to determine which classes should be grouped together in design.

Layers

Until this point in the development of our system, we have focused only on the problem domain; we have totally ignored the system environment (data management, user interface, and physical architecture). To successfully evolve the analysis model of the system into a design model of the system, we must add the system environment information. One useful way to do this, without overloading the developer, is to use *layers.* A layer represents an element of the software architecture of the evolving system. We have focused only on one layer in the evolving software architecture: the problem domain layer. There should be a layer for each of the different elements of the system environment (e.g., data management, user interface, physical architecture). Like partitions and collaborations, layers also can be portrayed using packages and package diagrams (see the next section of this chapter).

The idea of separating the different elements of the architecture into separate layers can be traced back to the MVC architecture of *Smalltalk.*[8] When Smalltalk was first created,[9] the authors decided to separate the application logic from the logic of the user interface. In this manner, it was possible to easily develop different user interfaces that worked with the same application. To accomplish this, they created the *Model–View–Controller (MVC)* architecture, where *Models* implemented the application logic (problem domain) and *Views* and *Controllers* implemented the logic for the user interface. Views handled the output, and Controllers handled the input. Because graphical user interfaces were first developed in the Smalltalk language, the MVC architecture served as the foundation for virtually all graphical user interfaces that have been developed today (including the Mac interfaces, the Windows family, and the various Unix-based GUI environments).

[8] See S. Lewis, *The Art and Science of Smalltalk: An Introduction to Object-Oriented Programming Using Visual-Works* (Englewood Cliffs, NJ: Prentice Hall, 1995).

[9] Smalltalk was invented in the early 1970s by a software-development research team at Xerox PARC. It introduced many new ideas into the area of programming languages (e.g., object orientation, windows-based user interfaces, reusable class library, and the development environment). In many ways, Smalltalk is the parent of all object-based and object-oriented languages, such as Visual Basic, C^{++}, and Java.

Layers	Examples	Relevant Chapters
Foundation	Date, Enumeration	7, 8
Problem Domain	Employee, Customer	4, 5, 6, 7, 8
Data Management	DataInputStream, FileInputStream	8, 9
Human–Computer Interaction	Button, Panel	8, 10
Physical Architecture	ServerSocket, URLConnection	8, 11

FIGURE 7-17
Layers and Sample Classes

Based on Smalltalk's innovative MVC architecture, many different software layers have been proposed.[10] We suggest the following layers on which to base software architecture: foundation, problem domain, data management, human–computer interaction, and physical architecture (see Figure 7-17). Each layer limits the types of classes that can exist on it (e.g., only user interface classes may exist on the human–computer interaction layer).

Foundation The *foundation layer* is, in many ways, a very uninteresting layer. It contains classes that are necessary for any object-oriented application to exist. They include classes that represent fundamental data types (e.g., integers, real numbers, characters, strings), classes that represent fundamental data structures, sometimes referred to as *container classes* (e.g., lists, trees, graphs, sets, stacks, queues), and classes that represent useful abstractions, sometimes referred to as *utility classes* (e.g., date, time, money). These classes are rarely, if ever, modified by a developer. They are simply used. Today, the classes found on this layer are typically included with the object-oriented development environments.

Problem Domain The *problem-domain layer* is what we have focused our attention on up until now. At this stage in the development of our system, we need to further detail the classes so that we can implement them in an effective and efficient manner. Many issues need to be addressed when designing classes, no matter on which layer they appear. For example, there are issues related to factoring, cohesion and coupling, connascence, encapsulation, proper use of inheritance and polymorphism, constraints, contract specification, and detailed method design. These issues are discussed in Chapter 8.

Data Management The *data management layer* addresses the issues involving the persistence of the objects contained in the system. The types of classes that appear in this layer deal with how objects can be stored and retrieved. The classes contained in this layer are called the Data Access and Manipulation (DAM) classes. The DAM classes allow the problem domain classes to be independent of the storage used and, hence, increase the portability of the evolving system. Some of the issues related to this layer include choice of the storage format and optimization. There is a plethora of different options in which to choose to store objects. These include sequential files, random access files, relational databases, object/relational databases, object-oriented databases,

[10] For example, Problem Domain, Human Interaction, Task Management, and Data Management [P. Coad and E. Yourdon, *Object-Oriented Design* (Englewood Cliffs, NJ: Yourdon Press, 1991)]; Domain, Application, and Interface (I. Graham, *Migrating to Object Technology* [Reading, MA: Addison-Wesley, 1994)]; Domain, Service, and Presentation [C. Larman, *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design* (Englewood Cliffs, NJ: Prentice Hall, 1998)]; Business, View, and Access [A. Bahrami, *Object-Oriented Systems Development using the Unified Modeling Language* (New York: McGraw-Hill, 1999)]; Application-Specific, Application-General, Middleware, System-Software [I. Jacobson, G. Booch, and J. Rumbaugh, *The Unified Software Development Process* (Reading, MA: Addison-Wesley, 1999)]; Foundation, Architecture, Business, and Application [M. Page-Jones, *Fundamentals of Object-Oriented Design in UML* (Reading, MA: Addison-Wesley, 2000)].

and NoSQL data stores. Each of these options has been optimized to provide solutions for different access and storage problems. Today, from a practical perspective, there is no single solution that optimally serves all applications. The correct solution is most likely some combination of the different storage options. A complete description of all the issues related to the *data management layer* is well beyond the scope of this book.[11] However, we do present the fundamentals in Chapter 9.

Human–Computer Interaction The *human–computer interaction layer* contains classes associated with the View and Controller idea from Smalltalk. The primary purpose of this layer is to keep the specific user-interface implementation separate from the problem domain classes. This increases the portability of the evolving system. Typical classes found on this layer include classes that can be used to represent buttons, windows, text fields, scroll bars, check boxes, drop-down lists, and many other classes that represent user-interface elements.

When designing the user interface for an application, many issues must be addressed: How important is consistency across different user interfaces? What about differing levels of user experience? How is the user expected to be able to navigate through the system? What about help systems and online manuals? What types of input elements should be included? What types of output elements should be included? Other questions that must be addressed are related to the platform on which the software will be deployed. For example, is the application going to run on a stand-alone computer, is it going to be distributed, or is the application going mobile? If it is expected to run on mobile devices, what type of platform: notebooks, tablets, or phones? Will it be deployed using Web technology, which runs on multiple devices, or will it be created using apps that are based on Android from Google, iOS from Apple, or Windows from Microsoft? Depending on the answer to these questions, different types of user interfaces are possible.

With the advent of social networking platforms, such as Facebook, Twitter, blogs, YouTube, and LinkedIn, the implications for the user interface can be mind boggling. Depending on the application, different social networking platforms may be appropriate for different aspects of the application. Furthermore, each of the different social networking platforms enables (or prevents) consideration of different types of user interfaces. Finally, with the potential audience of your application being global, many different cultural issues will arise in the design and development of culturally aware user interfaces (such as multilingual requirements). Obviously, a complete description of all the issues related to human–computer interaction is beyond the scope of this book.[12] However, from the user's perspective, the user interface is the system. We present the basic issues in user interface design in Chapter 10.

Physical Architecture The *physical architecture layer* addresses how the software will execute on specific computers and networks. This layer includes classes that deal with communication between the software and the computer's operating system and the network. For example, classes that address how to interact with the various ports on a specific computer are included in this layer.

[11] There are many good database design books that are relevant to this layer; see, for example, M. Gillenson, *Fundamentals of Database Management Systems* (Hoboken, NJ: John Wiley & Sons, 2005); F. R. McFadden, J. A. Hoffer, and Mary B. Prescott, *Modern Database Management,* 4th Ed. (Reading, MA: Addison-Wesley, 1998); M. Blaha and W. Premerlani, *Object-Oriented Modeling and Design for Database Applications* (Englewood Cliffs, NJ: Prentice Hall, 1998); R. J. Muller, *Database Design for Smarties: Using UML for Data Modeling* (San Francisco: Morgan Kaufmann, 1999).

[12] Books on user interface design that address these issues include B. Schheiderman, *Designing the User Interface: Strategies for Effective Human Computer Interaction*, 3rd Ed. (Reading, MA: Addison-Wesley, 1998); J. Tidwell, *Designing Interfaces: Patterns for Effective Interaction Design,* 2nd Ed. (Sebastopol, CA: O'Reilly Media, 2010); S. Krug, *Don't Make Me Think: A Common Sense Approach to Web Usability* (Berkeley, CA: New Riders Publishing, 2006); N. Singh and A. Pereira, *The Culturally Customized Web Site: Customizing Web Sites for the Global Marketplace* (Oxford, UK: Elsevier, 2005).

Unlike in the foundation layer, many design issues must be addressed before choosing the appropriate set of classes for this layer. These design issues include the choice of a computing or network architecture (such as the various client-server architectures), the actual design of a network, hardware and server software specification, and security issues. Other issues that must be addressed with the design of this layer include computer hardware and software configuration (choice of operating systems, such as Linux, Mac OSX, and Windows; processor types and speeds; amount of memory; data storage; and input/output technology), standardization, virtualization, grid computing, distributed computing, and Web services. This then leads us to one of the proverbial gorillas on the corner. What do you do with the cloud? The *cloud* is essentially a form of distributed computing. In this case, the cloud allows you to treat the platform, infrastructure, software, and even business processes as remote services that can be managed by another firm. In many ways, the cloud allows much of IT to be outsourced (see the discussion of outsourcing later in this chapter). Also as brought up with the human–computer interaction layer, the whole issue of mobile computing is very relevant to this layer. In particular, the different devices, such as phones and tablets, are relevant and the way they will communicate with each other, such as through cellular networks or WiFi, is also important.

Finally, given the amount of power that IT requires today, the whole topic of Green IT must be addressed. Topics that need to be addressed related to Green IT are the location of the data center, data center cooling, alternative power sources, reduction of consumables, the idea of a paperless office, Energy Star compliance, and the potential impact of virtualization, the cloud, and mobile computing. Like the data management and human–computer interaction layers, a complete description of all the issues related to the physical architecture is beyond the scope of this book.[13] However, we do present the basic issues in Chapter 11.

PACKAGES AND PACKAGE DIAGRAMS

In UML, collaborations, partitions, and layers can be represented by a higher-level construct: a package.[14] In fact, a package serves the same purpose as a folder on your computer. When packages are used in programming languages such as Java, packages are actually implemented as folders. A *package* is a general construct that can be applied to any of the elements in UML models. In Chapter 4, we introduced the idea of packages as a way to group use cases together to make the use-case diagrams easier to read and to keep the models at a reasonable level of complexity. In Chapters 5 and 6, we did the same thing for class and communication diagrams, respectively. In this section, we describe a *package diagram*: a diagram composed only of packages. A package diagram is effectively a class diagram that only shows packages.

The symbol for a package is similar to a tabbed folder (see Figure 7-18). Depending on where a package is used, packages can participate in different types of relationships. For example, in a class diagram, packages represent groupings of classes. Therefore, aggregation and association relationships are possible.

In a package diagram, it is useful to depict a new relationship, the *dependency relationship*. A dependency relationship is portrayed by a dashed arrow (see Figure 7-18). A dependency relationship represents the fact that a modification dependency exists between two packages. That is, it is possible that a change in one package could cause a change to

[13] Some books that cover these topics include S. D. Burd, *Systems Architecture,* 6th Ed. (Boston: Course Technology, 2011); I. Englander, *The Architecture of Computer Hardware, Systems Software, & Networking: An Information Technology Approach* (Hoboken, NJ: Wiley, 2009); K.K. Hausman and S. Cook, *IT Architecture for Dummies* (Hoboken, NJ: Wiley Publishing, 2011).

[14] This discussion is based on material in Chapter 7 of M. Fowler with K. Scott, *UML Distilled: A Brief Guide to the Standard Object Modeling Language,* 3rd Ed. (Reading, MA: Addison-Wesley, 2004).

A package: ■ Is a logical grouping of UML elements. ■ Is used to simplify UML diagrams by grouping related elements into a single higher-level element.	
A dependency relationship: ■ Represents a dependency between packages: If a package is changed, the dependent package also could have to be modified. ■ Has an arrow drawn from the dependent package toward the package on which it is dependent.	

FIGURE 7-18 Syntax for Package Diagram

be required in another package. Figure 7-19 portrays the dependencies among the different layers (foundation, problem domain, data management, human–computer interaction, and physical architecture). For example, if a change occurs in the problem domain layer, it most likely will cause changes to occur in the human–computer interaction, physical architecture, and data management layers. Notice that these layers point to the problem domain layer and therefore are dependent on it. However, the reverse is not true.[15] Also note that all layers are dependent upon the foundation layer. This is due to the contents of the foundation layer being the fundamental classes from which all other classes will be built. Consequently, any changes made to this layer could have ramificabitons to all other layers.

At the class level, there could be many causes for dependencies among classes. For example, if the protocol for a method is changed, then this causes the interface for all

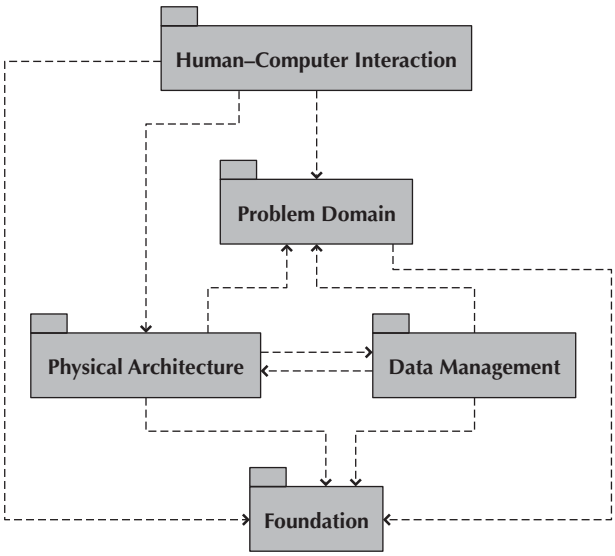


FIGURE 7-19
Package Diagram of Dependency Relationships among Layers

[15] A useful side effect of the dependencies among the layers is that the project manager can divide the project team up into separate subteams: one for each design layer. This is possible because each of the design layers is dependent on the problem domain layer, which has been the focus of analysis. In design, the team can gain some productivity-based efficiency by working on the different layer designs in parallel.

objects of this class to change. Therefore, all classes that have objects that send messages to the instances of the modified class might have to be modified. Capturing dependency relationships among the classes and packages helps the organization in maintaining object-oriented information systems.

Collaborations, partitions, and layers are modeled as packages in UML. Collaborations are normally factored into a set of partitions, which are typically placed on a layer. Partitions can be composed of other partitions. Also, it is possible to have classes in partitions, which are contained in another partition, which is placed on a layer. All these groupings are represented using packages in UML. Remember that a package is simply a generic grouping construct used to simplify UML models through the use of composition.[16]

A simple package diagram, based on the appointment system example from the previous chapters, is shown in Figure 7-20. This diagram portrays only a very small portion of the entire system. In this case, we see that the Patient UI, Patient-DAM, and Patient Table classes depend on the Patient class. Furthermore, the Patient-DAM class depends on the Patient Table class. The same can be seen with the classes dealing with the actual appointments. By isolating the Problem Domain classes (such as the Patient and Appt classes) from the actual object-persistence classes (such as the Patient Table and Appt Table classes) through the use of the intermediate Data Management classes (Patient-DAM and Appt-DAM classes), we isolate the Problem Domain classes from the actual storage medium.[17] This greatly simplifies the maintenance and increases the reusability of the Problem Domain classes. Of course, in a complete description of a real system, there would be many more dependencies.

Guidelines for Creating Package Diagrams

As with the UML diagrams described in the earlier chapters, we provide a set of guidelines that we have adapted from Ambler to create package diagrams.[18] In this case, we offer six guidelines.

- Use package diagrams to logically organize designs. Specifically, use packages to group classes together when there is an inheritance, aggregation, or composition relationship between them or when the classes form a collaboration.
- In some cases, inheritance, aggregation, or association relationships exist between packages. In those cases, for readability purposes, try to support inheritance relationships vertically, with the package containing the superclass being placed above the package containing the subclass. Use horizontal placement to support aggregation and association relationships, with the packages being placed side by side.
- When a dependency relationship exists on a diagram, it implies that there is at least one semantic relationship between elements of the two packages. The direction of the dependency is typically from the subclass to the superclass, from the whole to the part, and with contracts, from the client to the server. In other words, a subclass is dependent on the existence of a superclass, a whole is dependent upon its parts existing, and a client can't send a message to a nonexistent server.

[16] For those familiar with traditional approaches, such as structured analysis and design, packages serve a similar purpose as the leveling and balancing processes used in data flow diagramming.

[17] These issues are described in more detail in Chapter 9.

[18] S. W. Ambler, *The Elements of UML 2.0 Style* (Cambridge, UK: Cambridge University Press, 2005).

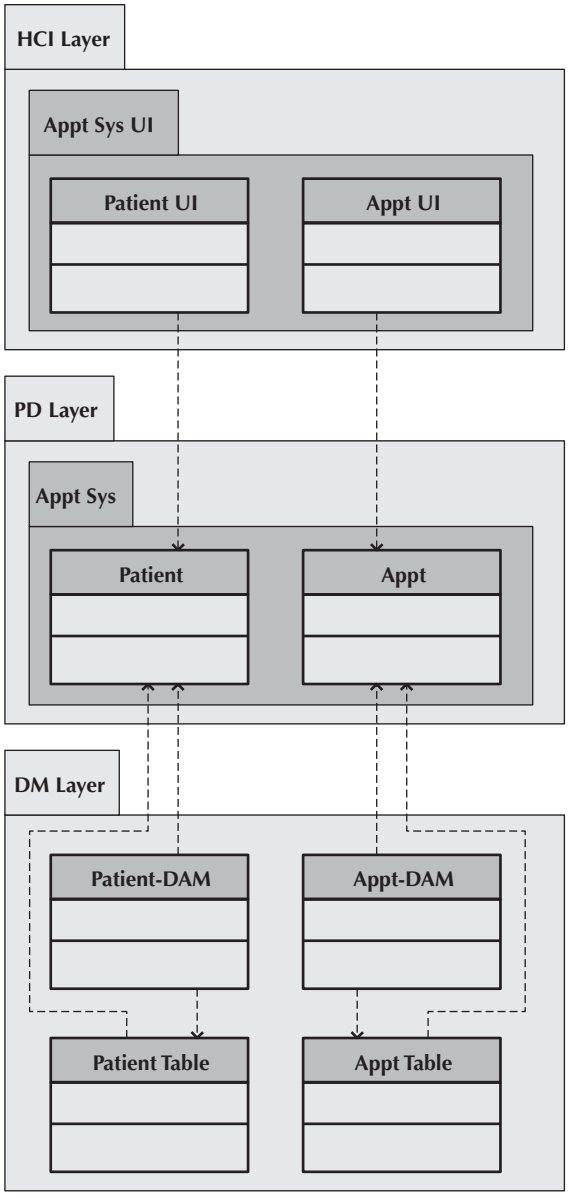
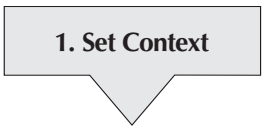


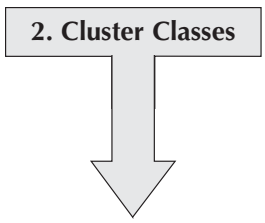
FIGURE 7-20
Partial Package Diagram of the Appointment System

- When using packages to group use cases together, be sure to include the actors and the associations that they have with the use cases grouped in the package. This will allow the diagram's user to better understand the context of the diagram.
- Give each package a simple, but descriptive name to provide the package diagram user with enough information to understand what the package encapsulates. Otherwise, the user will have to drill-down or open up the package to understand the package's purpose.
- Be sure that packages are cohesive. For a package to be cohesive, the classes contained in the package, in some sense, belong together. A simple, but not perfect, rule to follow when grouping classes together in a package is that the more the classes depend on each other, the more likely they belong together in a package.

Creating Package Diagrams

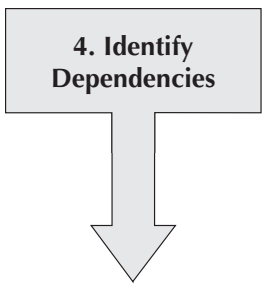


In this section, we describe a simple five-step process to create package diagrams. The first step is to set the context for the package diagram. Remember, packages can be used to model partitions and/or layers. Revisiting the appointment system again, let's set the context as the problem domain layer.

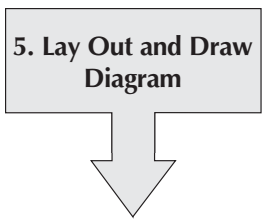


The second step is to cluster the classes together into partitions based on the relationships that the classes share. The relationships include generalization, aggregation, the various associations, and the message sending that takes place between the objects in the system. To identify the packages in the appointment system, we should look at the different analysis models [e.g., the class diagram (see Figure 7-15), the communication diagrams (see Figure 7-10)], and the CRUDE matrix (see Figure 7-12). Classes in a generalization hierarchy should be kept together in a single partition.

The third step is to place the clustered classes together in a partition and model the partitions as packages. Figure 7-21 portrays five packages in the PD Layer: Account Pkg, Participant Pkg, Patient Pkg, Appointment Pkg, and Treatment Pkg.



The fourth step is to identify the dependency relationships among the packages. We accomplish this by reviewing the relationships that cross the boundaries of the packages to uncover potential dependencies. In the appointment system, we see association relationships that connect the Account Pkg with the Appointment Pkg (via the associations between the Entry class and the Appointment class), the Participant Pkg with the Appointment Pkg (via the association between the Doctor class and the Appointment class), the Patient Pkg, which is contained within the Participant Pkg, with the Appointment Pkg (via the association between the Patient and Appointment classes), and the Patient Pkg with the Treatment Pkg (via the association between the Patient and Symptom classes).



The fifth step is to lay out and draw the diagram. Using the guidelines, place the packages and dependency relationships in the diagram. In the case of the Appointment system, there are dependency relationships between the Account Pkg and the Appointment Pkg, the Participant Pkg and the Appointment Pkg, the Patient Pkg and the Appointment Pkg, and the Patient Pkg and the Treatment Pkg. To increase the understandability of the dependency relationships among the different packages, a pure package diagram that shows only the dependency relationships among the packages can be created (see Figure 7-22).

Verifying and Validating Package Diagrams

Like all the previous models, package diagrams need to be verified and validated. In this case, the package diagrams were derived primarily from the class diagram, the communications diagrams, and the CRUDE matrix. Only two areas need to be reviewed.

First, the identified packages must make sense from a problem domain point of view. For example, in the context of an appointment system, the packages in Figure 7-22 (Participant, Patient, Appt, Account, and Treatment) seem to be reasonable.

Second, all dependency relationships must be based on message-sending relationships on the communications diagram, cell entries in the CRUDE matrix, and associations on the class diagram. In the case of the appointment system, the identified dependency relationships are reasonable (see Figures 7-10, 7-12, 7-15, and 7-22).

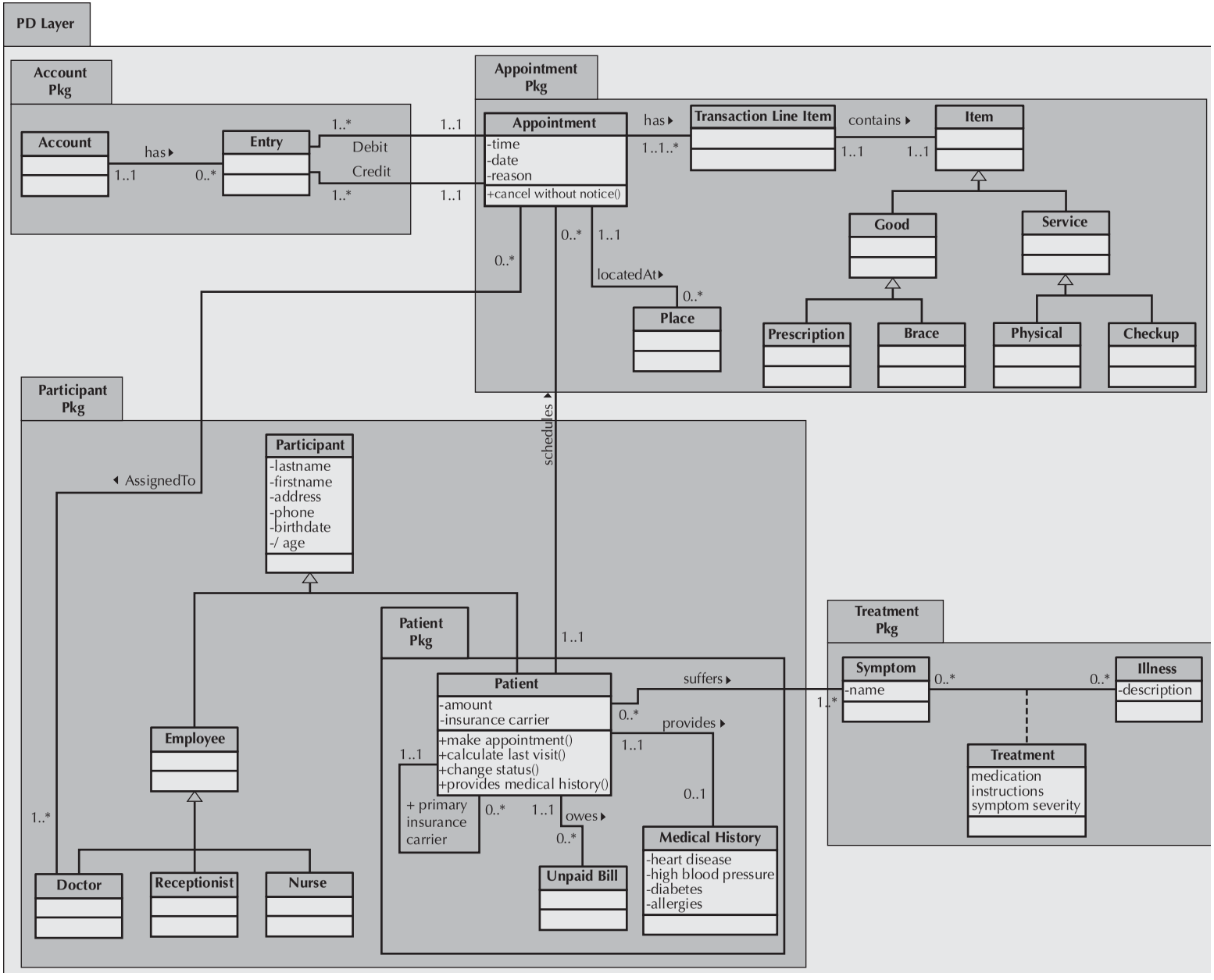


FIGURE 7-21 Package Diagram of the PD Layer of the Appointment Problem

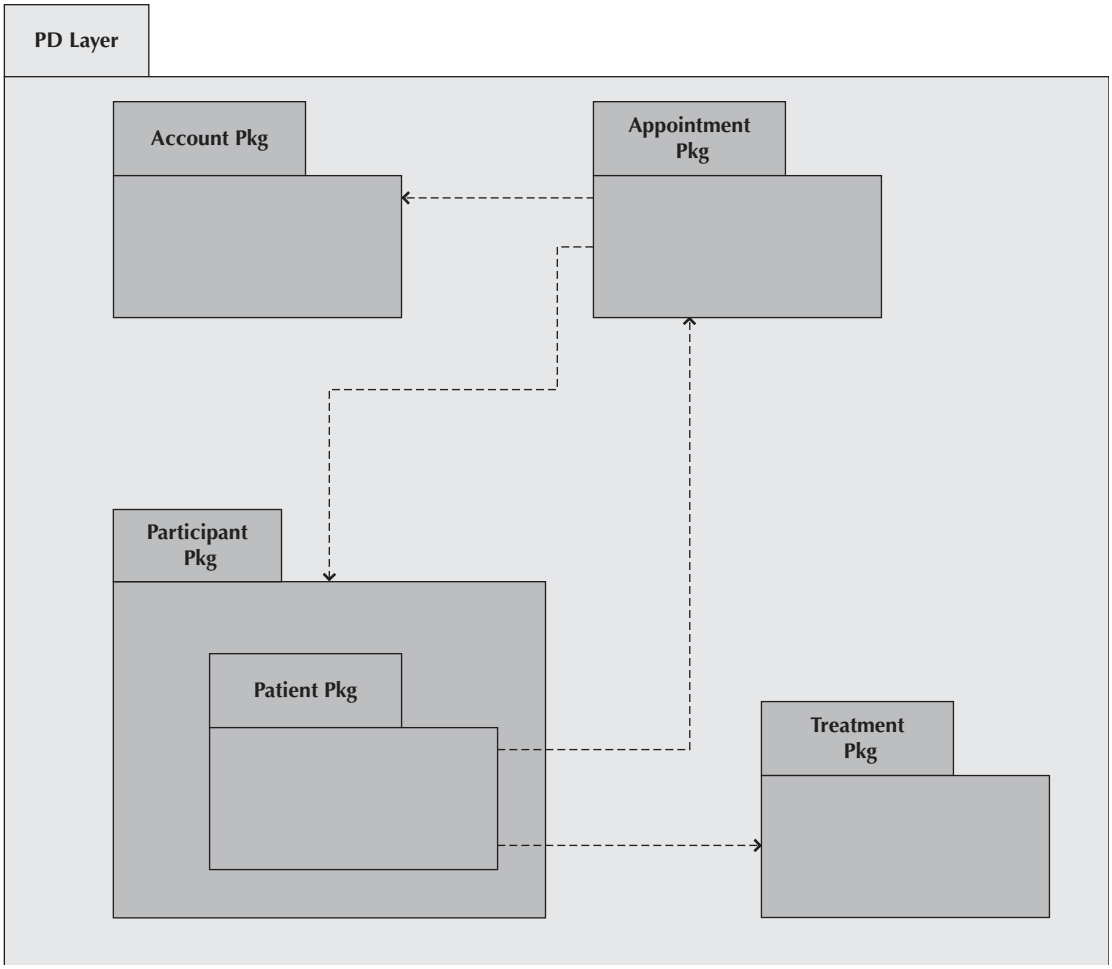


FIGURE 7-22 Overview Package Diagram of the PD Layer for the Appointment System

DESIGN STRATEGIES

Until now, we have assumed that the system will be built and implemented by the project team; however, there are actually three ways to approach the creation of a new system: developing a custom application in-house, buying and customizing a packaged system, and relying on an external vendor, developer, or service provider to build the system. Each of these choices has strengths and weaknesses, and each is more appropriate in different scenarios. The following sections describe each design choice in turn, and then we present criteria that you can use to select one of the three approaches for your project.

Custom Development

Many project teams assume that *custom development,* or building a new system from scratch, is the best way to create a system. For one thing, teams have complete control over the way the system looks and functions. Custom development also allows developers to be flexible and creative in the way they solve business problems. Additionally, a custom application is easier to change to include components that take advantage of current technologies that can support such strategic efforts.

Building a system in-house also builds technical skills and functional knowledge within the company. As developers work with business users, their understanding of the business grows and they become better able to align IS with strategies and needs. These same developers climb the technology learning curve so that future projects applying similar technology require much less effort.

Custom application development, however, requires dedicated effort that involves long hours and hard work. Many companies have a development staff who already is overcommitted to filling huge backlogs of systems requests and just does not have time for another project. Also, a variety of skills—technical, interpersonal, functional, project management, and modeling—must be in place for the project to move ahead smoothly. IS professionals, especially highly skilled individuals, are quite difficult to hire and retain.

The risks associated with building a system from the ground up can be quite high, and there is no guarantee that the project will succeed. Developers could be pulled away to work on other projects, technical obstacles could cause unexpected delays, and the business users could become impatient with a growing timeline.

Packaged Software

Many business needs are not unique, and because it makes little sense to reinvent the wheel, many organizations buy *packaged software* that has already been written rather than developing their own custom solution. In fact, there are thousands of commercially available software programs that have already been written to serve a multitude of purposes. Think about your own need for a word processor—did you ever consider writing your own word processing software? That would be very silly considering the number of good software packages available that are relatively inexpensive.

Similarly, most companies have needs that can be met quite well by packaged software, such as payroll or accounts receivable. It can be much more efficient to buy programs that have already been created, tested, and proven. Moreover, a packaged system can be bought and installed in a relatively short time when compared with a custom system. Plus, packaged systems incorporate the expertise and experience of the vendor who created the software.

Packaged software can range from reusable components to small, single-function tools to huge, all-encompassing systems such as *enterprise resource planning (ERP)* applications that are installed to automate an entire business. Implementing ERP systems is a process in which large organizations spend millions of dollars installing packages by companies such as SAP or Oracle and then change their businesses accordingly. Installing ERP software is much more difficult than installing small application packages because benefits can be harder to realize and problems are much more serious.

However, there are problems related to packaged software. For example, companies buying packaged systems must accept the functionality that is provided by the system, and rarely is there a perfect fit. If the packaged system is large in scope, its implementation could mean a substantial change in the way the company does business. Letting technology drive the business can be dangerous.

Most packaged applications allow *customization,* or the manipulation of system parameters to change the way certain features work. For example, the package might have a way to accept information about your company or the company logo that would then appear on input screens. Or an accounting software package could offer a choice of various ways to handle cash flow or inventory control so that it can support the accounting practices in different organizations. If the amount of customization is not enough and the software package has a few features that don't quite work the way the company needs it to work, the project team can create workarounds.

A *workaround* is a custom-built add-on program that interfaces with the packaged application to handle special needs. It can be a nice way to create needed functionality that does not exist in the software package. But workarounds should be a last resort for several reasons. First, workarounds are not supported by the vendor who supplied the packaged software, so upgrades to the main system might make the workaround ineffective. Also, if problems arise, vendors have a tendency to blame the workaround as the culprit and refuse to provide support.

Although choosing a packaged software system is simpler than custom development, it too can benefit from following a formal methodology, just as if a custom application were being built.

Systems integration refers to the process of building new systems by combining packaged software, existing legacy systems, and new software written to integrate these. Many consulting firms specialize in systems integration, so it is not uncommon for companies to select the packaged software option and then outsource the integration of a variety of packages to a consulting firm. (Outsourcing is discussed in the next section.)

The key challenge in systems integration is finding ways to integrate the data produced by the different packages and legacy systems. Integration often hinges on taking data produced by one package or system and reformatting it for use in another package or system. The project team starts by examining the data produced by and needed by the different packages or systems and identifying the transformations that must occur to move the data from one to the other. In many cases, this involves fooling the different packages or systems into thinking that the data were produced by an existing program module that the package or system expects to produce the data rather than the new package or system that is being integrated.

A third approach is through the use of an *object wrapper*.[19] An object wrapper is essentially an object that "wraps around" a legacy system, enabling an object-oriented system to send messages to the legacy system. Effectively, object wrappers create an application program interface (API) to the legacy system. The creation of an object wrapper protects the corporation's investment in the legacy system.

Outsourcing

The design choice that requires the least amount of in-house resources is *outsourcing*—hiring an external vendor, developer, or service provider to create the system. Outsourcing has become quite popular in recent years. Some estimate that as many as 50 percent of companies with IT budgets of more than $5 million are currently outsourcing or evaluating the approach.

With outsourcing, the decision making and/or management control of a business function is transferred to an outside supplier. This transfer requires two-way coordination, exchange of information, and trust between the supplier and the business. From an IT perspective, IT outsourcing can include hiring consultants to solve a specific problem, hiring contract programmers to implement a solution, hiring a firm to manage the IT function and assets of a company, or actually outsourcing the entire IT function to a separate firm. Today, through the use of application service providers (ASPs), Web services technology, and cloud services, it is possible to use a pay-as-you-go approach for a software package.[20] Essentially, IT outsourcing involves hiring a third party to perform some IT function that traditionally would be performed in-house.

There can be great benefit to having someone else develop a company's system. The outside company may be more experienced in the technology or have more resources, such as

[19] Ian Graham, *Object-Oriented Methods: Principles & Practice*, 3rd Ed. (Reading, MA: Addison-Wesley, 2001).

[20] For an economic explanation of how this could work, see H. Baetjer, *Software as Capital: An Economic Perspective on Software Engineering* (Los Alamitos, CA: IEEE Computer Society Press, 1997).

experienced programmers. Many companies embark upon outsourcing deals to reduce costs, whereas others see it as an opportunity to add value to the business.

For whatever reason, outsourcing can be a good alternative for a new system. However, it does not come without costs. If you decide to leave the creation of a new system in the hands of someone else, you could compromise confidential information or lose control over future development. In-house professionals are not benefiting from the skills that could be learned from the project; instead, the expertise is transferred to the outside organization. Ultimately, important skills can walk right out the door at the end of the contract. Furthermore, when offshore outsourcing is being considered, we must also be cognizant of language issues, time-zone differences, and cultural differences (e.g., acceptable business practices as understood in one country that may be unacceptable in another). All these concerns, if not dealt with properly, can prevail over any advantage that outsourcing or offshore outsourcing could realize.

Most risks can be addressed if a company decides to outsource, but two are particularly important. First, the company must thoroughly assess the requirements for the project—a company should never outsource what is not understood. If rigorous planning and analysis has occurred, then the company should be well aware of its needs. Second, the company should carefully choose a vendor, developer, or service with a proven track record with the type of system and technology that its system needs.

Three primary types of contracts can be drawn to control the outsourcing deal. A *time-and-arrangements contract* is very flexible because a company agrees to pay for whatever time and expenses are needed to get the job done. Of course, this agreement could result in a large bill that exceeds initial estimates. This works best when the company and the outsourcer are unclear about what it is going to take to finish the job.

A company will pay no more than expected with a *fixed-price contract* because if the outsourcer exceeds the agreed-upon price, it will have to absorb the costs. Outsourcers are much more careful about defining requirements clearly up front, and there is little flexibility for change.

The type of contract gaining in popularity is the *value-added contract,* whereby the outsourcer reaps some percentage of the completed system's benefits. The company has very little risk in this case, but it must expect to share the wealth once the system is in place.

Creating fair contracts is an art because flexibility must be carefully balanced with clearly defined terms. Often, needs change over time. Therefore, the contract should not be so specific and rigid that alterations cannot be made. Think about how quickly mobile technology has changed. It is difficult to foresee how a project might evolve over a long period of time. Short-term contracts help leave room for reassessment if needs change or if relationships are not working out the way both parties expected. In all cases, the relationship with the outsourcer should be viewed as a partnership where both parties benefit and communicate openly.

Managing the outsourcing relationship is a full-time job. Thus, someone needs to be assigned full time to manage the outsourcer, and the level of that person should be appropriate for the size of the job (a multimillion dollar outsourcing engagement should be handled by a high-level executive). Throughout the relationship, progress should be tracked and measured against predetermined goals. If a company does embark upon an outsourcing design strategy, it should be sure to get adequate information. Many books have been written that provide much more detailed information on the topic.[21] Figure 7-23 summarizes some guidelines for outsourcing.

[21] For more information on outsourcing, we recommend M. Lacity and R. Hirschheim, *Information Systems Outsourcing: Myths, Metaphors, and Realities* (New York, NY: Wiley, 1993); L. Willcocks and G. Fitzgerald, *A Business Guide to Outsourcing Information Technology* (London: Business Intelligence, 1994); E. Carmel, *Offshoring Information Technology: Sourcing and Outsourcing to a Global Workforce* (Cambridge, England: Cambridge University Press, 2005); J. K. Halvey and B. M. Melby, *Information Technology Outsourcing Transactions: Process, Strategies, and Contracts,* 2nd Ed. (Hoboken, NJ: Wiley, 2005); T. L. Friedman, *The World Is Flat: A Brief History of the Twenty-First Century, Updated and Expanded Edition* (New York: Farrar, Straus, and Giroux, 2006).

Outsourcing
• Keep the lines of communication open between you and your outsourcer. • Define and stabilize requirements before signing a contact. • View the outsourcing relationship as a partnership. • Select the vendor, developer or service provider carefully. • Assign a person to managing the relationship. • Don't outsource what you don't understand. • Emphasize flexible requirements, long-term relationships and short-term contracts.

FIGURE 7-23
Outsourcing Guidelines

Selecting a Design Strategy

Each of the design strategies just discussed has its strengths and weaknesses, and no one strategy is inherently better than the others. Thus, it is important to understand the strengths and weaknesses of each strategy and when to use each. Figure 7-24 summarizes the characteristics of each strategy.

Business Need If the business need for the system is common and technical solutions already exist that can meet the business need of the system, it makes little sense to build a custom application. Packaged systems are good alternatives for common business needs. A custom alternative should be explored when the business need is unique or has special requirements. Usually, if the business need is not critical to the company, then outsourcing is the best choice—someone outside of the organization can be responsible for the application development.

In-house Experience If in-house experience exists for all the functional and technical needs of the system, it will be easier to build a custom application than if these skills do not exist. A packaged system may be a better alternative for companies that do not have the technical skills to build the desired system. For example, a project team that does not have mobile technology skills might want to consider outsourcing those aspects of the system.

Project Skills The skills that are applied during projects are either technical (e.g., Java, SQL) or functional (e.g., security), and different design alternatives are more viable, depending on

	Use Custom Development When…	Use a Packaged System When…	Use Outsourcing When…
Business Need	The business need is unique.	The business need is common.	The business need is not core to the business.
In-house Experience	In-house functional and technical experience exists.	In-house functional experience exists.	In-house functional or technical experience does not exist.
Project Skills	There is a desire to build in-house skills.	The skills are not strategic.	The decision to outsource is a strategic decision.
Project Management	The project has a highly skilled project manager and a proven methodology.	The project has a project manager who can coordinate the vendor's efforts.	The project has a highly skilled project manager at the level of the organization that matches the scope of the outsourcing deal.
Time frame	The time frame is flexible.	The time frame is short.	The time frame is short or flexible.

FIGURE 7-24 Selecting a Design Strategy

how important the skills are to the company's strategy. For example, if certain functional and technical expertise that relates to mobile application development is important to an organization because it expects mobile to play an important role in its sales over time, then it makes sense for the company to develop mobile applications in-house, using company employees so that the skills can be developed and improved. On the other hand, some skills, such as network security, may be beyond the technical expertise of employees or not of interest to the company's strategists—it is just an operational issue that needs to be addressed. In this case, packaged systems or outsourcing should be considered so that internal employees can focus on other business-critical applications and skills.

Project Management Custom applications require excellent project management and a proven methodology. So many things, such as funding obstacles, staffing holdups, and overly demanding business users, can push a project off-track. Therefore, the project team should choose to develop a custom application only if it is certain that the underlying coordination and control mechanisms will be in place. Packaged and outsourcing alternatives also need to be managed; however, they are more shielded from internal obstacles because the external parties have their own objectives and priorities (e.g., it may be easier for an outside contractor to say no to a user than it is for a person within the company). Typically, packaged and outsourcing alternatives have their own methodologies, which can benefit companies that do not have an appropriate methodology to use.

Time Frame When time is a factor, the project team should probably start looking for a system that is already built and tested. In this way, the company will have a good idea of how long the package will take to put in place and what the final result will contain. The time frame for custom applications is hard to pin down, especially when you consider how many projects end up missing important deadlines. If a company must choose the custom development alternative and the time frame is very short, it should consider using techniques such as timeboxing to manage this problem. The time to produce a system using outsourcing really depends on the system and the outsourcer's resources. If a service provider has services in place that can be used to support the company's needs, then a business need could be implemented quickly. Otherwise, an outsourcing solution could take as long as a custom development initiative.

SELECTING AN ACQUISITION STRATEGY

Once the project team has a good understanding of how well each design strategy fits with the project's needs, it must begin to understand exactly *how* to implement these strategies. For example, what tools and technology would be used if a custom alternative were selected? What vendors make packaged systems that address the project's needs? What service providers would be able to build this system if the application were outsourced? This information can be obtained from people working in the IS department and from recommendations by business users. Alternatively, the project team can contact other companies with similar needs and investigate the types of systems that they have put in place. Vendors and consultants usually are willing to provide information about various tools and solutions in the form of brochures, product demonstrations, and information seminars. However, a company should be sure to validate the information it receives from vendors and consultants. After all, they are trying to make a sale. Therefore, they may stretch the capabilities of their tool by focusing on only the positive aspects of the tool while omitting the tool's drawbacks.

It is likely that the project team will identify several ways that a system could be constructed after weighing the specific design options. For example, the project team might have

found three vendors that make packaged systems that potentially could meet the project's needs. Or the team may be debating over whether to develop a system using Java as a development tool and the database management system from Oracle or to outsource the development effort to a consulting firm such as Accenture or CGI. Each alternative has pros and cons associated with it that need to be considered, and only one solution can be selected in the end.

To aid in this decision, additional information should be collected. Project teams employ several approaches to gather additional information that is needed. One helpful tool is the request for proposal (RFP), a document that solicits a formal proposal from a potential vendor, developer, or service provider. RFPs describe in detail the system or service that is needed, and vendors respond by describing in detail how they could supply those needs.

Although there is no standard way of writing an RFP, it should include certain key facts that the vendor requires, such as a detailed description of needs, any special technical needs or circumstances, evaluation criteria, procedures to follow, and a timetable. In a large project, the RFP can be hundreds of pages long, since it is essential that all required project details are included.

The RFP is not just a way to gather information. Rather, it results in a vendor proposal that is a binding offer to accomplish the tasks described in the RFP. The vendor proposal includes a schedule and a price for which the work is to be performed. Once the winning vendor proposal is chosen, a contract for the work is developed and signed by both parties.

For smaller projects with smaller budgets, the request for information (RFI) may be sufficient. An RFI is a shorter, less detailed request that is sent to potential vendors to obtain general information about their products and services. Sometimes, the RFI is used to determine which vendors have the capability to perform a service. It is often then followed up with an RFP to the qualified vendors.

When a list of equipment is so complete that the vendor need only provide a price, without any analysis or description of what is needed, the request for quote (RFQ) may be used. For example, if twenty long-range RFID tag readers are needed from the manufacturer on a certain date at a certain location, the RFQ can be used. If an item is described, but a specific manufacturer's product is not named, then extensive testing will be required to verify fulfillment of the specifications.

Alternative Matrix

An *alternative matrix* can be used to organize the pros and cons of the design alternatives so that the best solution will be chosen in the end (see Figure 7-25). This matrix is created using the same steps as the feasibility analysis, which was presented in Chapter 2. The only difference is that the alternative matrix combines several feasibility analyses into one matrix so that the alternatives can easily be compared. An alternative matrix is a grid that contains the technical, budget, and organizational feasibilities for each system candidate, pros and cons associated with adopting each solution, and other information that is helpful when making comparisons. Sometimes weights are provided for different parts of the matrix to show when some criteria are more important to the final decision.

To create the alternative matrix, draw a grid with the alternatives across the top and different criteria (e.g., feasibilities, pros, cons, and other miscellaneous criteria) along the side. Next, fill in the grid with detailed descriptions about each alternative. This becomes a useful document for discussion because it clearly presents the alternatives being reviewed and comparable characteristics for each one.

Sometimes, weights and scores are added to the alternative matrix to create a weighted alternative matrix that communicates the project's most important criteria and the alternatives that best address them. A scorecard is built by adding a column labeled "weight" that includes

a number depicting how much each criterion matters to the final decision. Typically, analysts take 100 points and spread them out across the criteria appropriately. If five criteria were used and all mattered equally, then each criterion would receive a weight of 20. However, if costs were the most important criterion for choosing an alternative, it might receive 60 points, and the other four criteria might get only 10 points each.

Then, the analysts add to the matrix a column called "Score" that communicates how well each alternative meets the criteria. Usually, number ranges like 1 to 5 or 1 to 10 are used to rate the appropriateness of the alternatives by the criteria. So, for the cost criterion, the least expensive alternative may receive a 5 on a 1-to-5 scale, whereas a costly alternative would receive a 1. Weighted scores are computed with each criterion's weight multiplied by the score it was given for each alternative. Then, the weighted scores are totaled for each alternative. The highest weighted score achieves the best match for our criteria. When numbers are used in the alternative matrix, project teams can make decisions quantitatively and on the basis of hard numbers.

It should be pointed out, however, that the score assigned to the criteria for each alternative is nothing more than a subjective assignment. Consequently, it is entirely possible for an analyst to skew the analysis according to his or her own biases. In other words, the weighted alternative matrix can be made to support whichever alternative you prefer and yet retains the appearance of an objective, rational analysis. To avoid the problem of a biased analysis, each analyst on the team could develop ratings independently; then, the ratings could be compared and discrepancies resolved in an open team discussion.

The final step, of course, is to decide which solution to design and implement. The decision should be made by a combination of business users and technical professionals after the issues involved with the different alternatives are well understood. Once the decision is finalized, design can continue as needed, based on the selected alternative.

Evaluation Criteria	Relative Importance (Weight)	Alternative 1: Custom Application Using VB.NET	Score (1–5)*	Weighted Score	Alternative 2: Custom Application Using Java	Score (1–5)*	Weighted Score	Alternative 3: Packaged Software Product ABC	Score (1–5)*	Weighted Score
Technical Issues:										
Criterion 1	20		5	100		3	60		3	60
Criterion 2	10		3	30		3	30		5	50
Criterion 3	10		2	20		1	10		3	30
Economic Issues:										
Criterion 4	25	Supporting	3	75	Supporting	3	75	Supporting	5	125
Criterion 5	10	Information	3	30	Information	1	10	Information	5	50
Organizational Issues:										
Criterion 6	10		5	50		5	50		3	30
Criterion 7	10		3	30		3	30		1	10
Criterion 8	5		3	15		1	5		1	5
TOTAL	100			350			270			360

* This denotes how well the alternative meets the criteria. 1 = poor fit; 5 = perfect fit.

FIGURE 7-25 Sample Alternative Matrix Using Weights

APPLYING THE CONCEPTS AT PATTERSON SUPERSTORE

The team had one major task to complete before moving into design. After developing and verifying the functional, structural, and behavioral models, they now had to validate that the functional, structural, and behavioral models developed in analysis agreed with each other. In other words, they needed to balance the functional, structural, and behavioral models. As you will see, this activity revealed inconsistencies and uncovered new information about the system that they hope to implement. After creating corrected iterations of each of the three types of models, the team explored design alternatives and determined a design strategy.

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 balancing the analysis models.
- ☐ Balance the functional models with the structural models.
- ☐ Balance the functional models with the behavioral models.
- ☐ Balance the structural models with the behavioral models.
- ☐ Describe the purpose of the factoring, refinement, and abstraction processes.
- ☐ Describe the purpose of partitions and collaborations.
- ☐ Name and describe the layers.
- ☐ Explain the purpose of a package diagram.
- ☐ Describe the different elements of the package diagram.
- ☐ Create a package diagram to model partitions and layers.
- ☐ Verify and validate package diagrams using walkthroughs.
- ☐ Describe the pros and cons of the three basic design strategies.
- ☐ Describe the basis of selecting a design strategy.
- ☐ Explain how and when to use RFPs, RFIs, and RFQs to gather information from vendors.
- ☐ Describe how to use a weighted alternative matrix to select an acquisition strategy.

KEY TERMS

A-kind-of
Abstract classes
Abstraction
Aggregation
Alternative matrix
Balancing the models
Class
Client
Collaboration
Concrete classes
Contract
Controller
Custom development
Customization
Data management layer
Dependency relationship
Enterprise resource systems (ERP)
Factoring
Fixed-price contract
Foundation layer
Generalization
Has-parts
Human–computer interaction layer
Layer
Message
Method
Model
Model-View-Controller (MVC)
Module
Object wrapper
Outsourcing
Package
Package diagram
Packaged software
Partition
Physical architecture layer
Problem domain layer
Refinement
Request for information (RFI)
Request for proposals (RFP)
Server
Smalltalk
Systems integration
Time-and-arrangements contract
Validation
Value-added contract
Verification
View
Workaround

QUESTIONS

1. Explain the primary difference between an analysis model and a design model.
2. What is meant by balancing the models?
3. What are the interrelationships among the functional, structural, and behavioral models that need to be tested?
4. What does factoring mean? How is it related to abstraction and refinement?
5. What is a partition? How does a partition relate to a collaboration?
6. What is a layer? Name the different layers.
7. What is the purpose of the different layers?
8. Describe the different types of classes that can appear on each of the layers.
9. What issues or questions arise on each of the different layers?
10. What is a package? How are packages related to partitions and layers?
11. What is a dependency relationship? How do you identify them?
12. What are the five steps for identifying packages and creating package diagrams?
13. What needs to be verified and validated in package diagrams?
14. When drawing package diagrams, what guidelines should you follow?
15. What situations are most appropriate for a custom development design strategy?
16. What are some problems with using a packaged software approach to building a new system? How can these problems be addressed?
17. Why do companies invest in ERP systems?
18. What are the pros and cons of using a workaround?
19. When is outsourcing considered a good design strategy? When is it not appropriate?
20. What is an object wrapper?
21. What is systems integration? Explain the challenges.
22. What are the differences between the time-and-arrangements, fixed-price, and value-added contracts for outsourcing?
23. How are the alternative matrix and feasibility analysis related?
24. What is an RFP? How is this different from an RFI?

EXERCISES

A. For the A Real Estate Inc. problem in Chapters 4 (exercises I, J, and K), 5 (exercises P and Q), and 6 (exercise D):

1. Perform a verification and validation walkthrough of the functional, structural, and behavioral models to ensure that all between-model issues have been resolved.
2. Using the communication diagrams and the CRUDE matrix, create a package diagram of the problem domain layer.
3. Perform a verification and validation walkthrough of the package diagram.
4. Based on the analysis models that have been created and your current understanding of the firm's position, what design strategy would you recommend? Why?

B. For the A Video Store problem in Chapters 4 (exercises L, M, and N), 5 (exercises R and S), and 6 (exercise E):

1. Perform a verification and validation walkthrough of the functional, structural, and behavioral models to ensure that all between-model issues have been resolved.
2. Using the communication diagrams and the CRUDE matrix, create a package diagram of the problem domain layer.
3. Perform a verification and validation walkthrough of the package diagram.
4. Based on the analysis models that have been created and your current understanding of the firm's position, what design strategy would you recommend? Why?

C. For the health club membership problem in Chapters 4 (exercises O, P, and Q), 5 (exercises T and U), and 6 (exercise F):

1. Perform a verification and validation walkthrough of the functional, structural, and behavioral models to ensure that all between-model issues have been resolved.
2. Using the communication diagrams and the CRUDE matrix, create a package diagram of the problem domain layer.
3. Perform a verification and validation walkthrough of the package diagram.

4. Based on the analysis models that have been created and your current understanding of the firm's position, what design strategy would you recommend? Why?

D. For the Picnics R Us problem in Chapters 4 (exercises R, S, and T), 5 (exercises V and W), and 6 (exercise G):

1. Perform a verification and validation walkthrough of the functional, structural, and behavioral models to ensure that all between-model issues have been resolved.
2. Using the communication diagrams and the CRUDE matrix, create a package diagram of the problem domain layer.
3. Perform a verification and validation walkthrough of the package diagram.
4. Based on the analysis models that have been created and your current understanding of the firm's position, what design strategy would you recommend? Why?

E. For the Of-the-Month-Club problem in Chapters 4 (exercises U, V, and W), 5 (exercises X and Y), and 6 (exercise H):

1. Perform a verification and validation walkthrough of the functional, structural, and behavioral models to ensure that all between-model issues have been resolved.
2. Using the communication diagrams and the CRUDE matrix, create a package diagram of the problem domain layer.
3. Perform a verification and validation walkthrough of the package diagram.
4. Based on the analysis models that have been created and your current understanding of the firm's position, what design strategy would you recommend? Why?

F. Suppose you are leading a project that will implement a new course-enrollment system for your university. You are thinking about either using a packaged course-enrollment application or outsourcing the job to an external consultant. Create an outline for an RFP to which interested vendors and consultants could respond.

G. Suppose you and your friends are starting a small business painting houses in the summertime. You need to buy a software package that handles the financial transactions of the business. Create an alternative matrix that compares three packaged systems (e.g., Quicken, MS Money, Quickbooks). Which alternative appears to be the best choice?

MINICASES

1. Susan, president of MOTO, Inc., a human resources management firm, is reflecting on the client management software system her organization purchased four years ago. At that time, the firm had just gone through a major growth spurt, and the mixture of automated and manual procedures that had been used to manage client accounts became unwieldy. Susan and Nancy, her IS department head, researched and selected the package that is currently used. Susan had heard about the software at a professional conference she attended, and, at least initially, it worked fairly well for the firm. Some of their procedures had to change to fit the package, but they expected that and were prepared for it.

   Since that time, MOTO, Inc., has continued to grow, not only through an expansion of the client base but also through the acquisition of several smaller employment-related businesses. MOTO, Inc., is a much different business than it was four years ago. Along with expanding to offer more diversified human resources management services, the firm's support staff has also expanded. Susan and Nancy are particularly proud of the IS department they have built up over the years. Using strong ties with a local university, an attractive compensation package, and a good working environment, the IS department is well staffed with competent, innovative people, plus a steady stream of college interns that keeps the department fresh and lively. One of the IS teams pioneered the use of the Internet to offer MOTO's services to a whole new market segment, an experiment that has proved very successful.

   It seems clear that a major change is needed in the client-management software, and Susan has already begun to plan financially to undertake such a project. This software is a central part of MOTO's operations, and Susan wants to be sure that a high-quality system is obtained this time. She knows that the vendor of their current system has made some revisions and additions to its product line. A number of other

software vendors also offer products that may be suitable. Some of these vendors did not exist when the purchase was made four years ago. Susan is also considering Nancy's suggestion that the IS department develop a custom software application.

a. Outline the issues that Susan should consider that would support the development of a custom software application in-house.

b. Outline the issues that Susan should consider that would support the purchase of a software package.

c. Within the context of a systems-development project, when should the decision of make-versus-buy be made? How should Susan proceed? Explain your answer.

2. Refer to minicase 1 (West Star Marinas) in Chapter 5. After all the analysis models (both the as-is and to-be models) for West Star Marinas were completed, the director of operations finally understood why it was important to understand the as-is system before delving into the development of the to-be system. However, you now tell him that the to-be models are only the problem-domain portion of the design. He is now very confused. After explaining to him the advantages of using a layered approach to developing the system, he says, "I don't care about reusability or maintenance. I only want the system to be implemented as soon as possible. You IS types are always trying to pull a fast one on the users. Just get the system completed."

What is your response to the Director of Operations? Do you jump into implementation as he seems to want? What do you do next?

3. Refer to the analysis models that you created for professional and scientific staff management (PSSM) for minicase 2 in Chapter 4 and for minicase 1 in Chapter 6.

a. Perform a verification and validation walkthrough of the functional, structural, and behavioral models to ensure that all between-model issues have been resolved.

b. Using the communication diagrams and the CRUDE matrix, create a package diagram of the problem domain layer.

c. Perform a verification and validation walkthrough of the package diagram.

d. Based on the analysis models that have been created and your current understanding of the firm's position, what design strategy would you recommend? Why?

4. Refer to the analysis models that you created for Holiday Travel Vehicles for minicase 2 in Chapter 5 and for minicase 2 in Chapter 6.

a. Perform a verification and validation walkthrough of the functional, structural, and behavioral models to ensure that all between-model issues have been resolved.

b. Using the communication diagrams and the CRUDE matrix, create a package diagram of the problem domain layer.

c. Perform a verification and validation walkthrough of the package diagram.

d. Based on the analysis models that have been created and your current understanding of the firm's position, what design strategy would you recommend? Why?

CHAPTER 8

Class and Method Design

The most important step of the design phase is designing the individual classes and methods. Object-oriented systems can be quite complex, so analysts need to create instructions and guidelines for programmers that clearly describe what the system must do. This chapter presents a set of criteria, activities, and techniques used to design classes and methods. Together they are used to ensure that the object-oriented design communicates how the system needs to be coded.

OBJECTIVES

- Become familiar with coupling, cohesion, and connascence.
- Be able to specify, restructure, and optimize object designs.
- Be able to identify the reuse of predefined classes, libraries, frameworks, and components.
- Be able to specify constraints and contracts.
- Be able to create a method specification.

INTRODUCTION

WARNING: *This material may be hazardous to your mental stability*. Not really, but now that we have your attention, you must realize that this material is fairly technical in nature and that it is extremely important in today's "flat" world. Today, much of the actual implementation will be done in a different geographic location than where the analysis and design are performed. We must ensure that the design is specified in a "correct" manner and that there is no, or at least minimal, ambiguity in the design specification.

In today's flat world, the common language spoken among developers is very likely to be UML and some object-oriented language, such as Java, and not English. English has always been and always will be ambiguous. Furthermore, to what variety of English do we refer? As both Oscar Wilde and George Bernard Shaw independently pointed out, the United States and England are divided by a common language.

Practically speaking, Class and Method design is where all the work actually gets done during design. No matter which layer you are focusing on, the classes, which will be used to create the system objects, must be designed. Some people believe that with reusable class libraries and off-the-shelf components, this type of low-level, or detailed, design is a waste of time and that we should jump immediately into the "real" work: coding the system. However, past experience shows that low-level, or detailed, design is critical despite the use of libraries and components. Detailed design is still very important for three reasons. First, with today's modern CASE tools, quite a bit of the actual code can be generated by the tool from the detailed design. Second, even preexisting classes and components need to be understood, organized, and pieced together. Third, it is still common for the project team

to have to write some code and produce original classes that support the application logic of the system.

Jumping right into coding will guarantee disastrous results. For example, even though the use of layers can simplify the individual classes, they can increase the complexity of the interactions between them. If the classes are not designed carefully, the resulting system can be very inefficient. Or worse, the instances of the classes (i.e., the objects) will not be capable of communicating with each other, which will result in the system's not working properly.

In an object-oriented system, changes can take place at different levels of abstraction. These levels include variable, method, class/object, package,[1] library, and/or application/system levels (see Figure 8-1). The changes that take place at one level can affect other levels (e.g., changes to a class can affect the package level, which can affect both the system level and the library level, which in turn can cause changes back down at the class level). Finally, changes can occur at different levels at the same time.

The good news is that the detailed design of the individual classes and methods is fairly straightforward. The interactions among the objects on the problem-domain layer have been designed, in some detail, during analysis (see Chapters 4 through 6). The other layers (data management, human–computer interaction, and physical architecture) are highly dependent on the problem-domain layer. Therefore, if the problem-domain classes are designed correctly, the design of the classes on the other layers will fall into place, relatively speaking.

That being said, it has been our experience that many project teams are much too quick at jumping into writing code for the classes without first designing them. Some of this has been caused by the fact that object-oriented systems analysis and design has evolved from object-oriented programming. Until recently there has been a general lack of accepted guidelines on how to design and develop effective object-oriented systems. However, with the

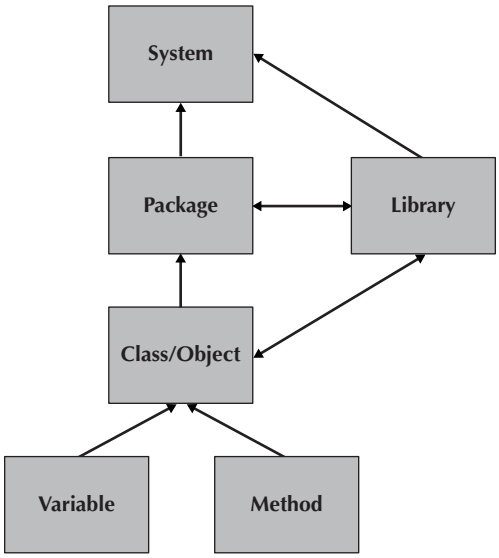


FIGURE 8-1
Levels of Abstraction in Object-Oriented Systems

Source: Based on material from David P. Tegarden, Steven D. Sheetz, and David E. Monarchi, "A Software Complexity Model of Object-Oriented Systems," *Decision Support Systems* 13 (March 1995): 241–262.

[1] A package is a group of collaborating objects. Other names for a package include cluster, partition, pattern, subject, and subsystem.

acceptance of UML as a standard object notation, standardized approaches based on work of many object methodologists have begun to emerge.[2]

REVIEW OF THE BASIC CHARACTERISTICS OF OBJECT ORIENTATION

Object-oriented systems can be traced back to the Simula and the Smalltalk programming languages. However, until the increase in processor power and the decrease in processor cost that occurred in the 1980s, object-oriented approaches were not practical. Many of the specific details concerning the basic characteristics of object-orientation are language dependent; that is, each object-oriented programming language tends to implement some of the object-oriented basics in a different way. Consequently, we need to know which programming language is going to be used to implement the different aspects of the solution. Otherwise, the system could behave in a manner different than the analyst, designer, and client expect. Today, the C++, Java, Objective-C, and Visual Basic programming languages tend to be the more predominant languages used. In this section, we review the basic characteristics of object orientation and point out where the language-specific issues emerge.

Classes, Objects, Methods, and Messages

The basic building block of the system is the *object*. Objects are *instances* of classes. *Classes* are templates that we use to define both the data and processes that each object contains. Each object has *attributes* that describe data about the object. Objects have *state,* which is defined by the value of its attributes and its relationships with other objects at a particular point in time. And each object has *methods,* which specify what processes the object can perform. From our perspective, methods are used to implement the *operations* that specified the *behavior* of the objects (see Chapter 5). To get an object to perform a method (e.g., to delete itself), a *message* is sent to the object. A message is essentially a function or procedure call from one object to another object.

Encapsulation and Information Hiding

Encapsulation is the mechanism that combines the processes and data into a single object. *Information hiding* suggests that only the information required to use an object be available outside the object; that is, information hiding is related to the *visibility* of the methods and attributes (see Chapter 5). Exactly how the object stores data or performs methods is not relevant, as long as the object functions correctly. All that is required to use an object are the set of methods and the messages needed to be sent to trigger them. The only communication between objects should be through an object's methods. The fact that we can use an object by sending a message that calls methods is the key to reusability because it shields the internal workings of the object from changes in the outside system, and it keeps the system from being affected when changes are made to an object.

Polymorphism and Dynamic Binding

Polymorphism means having the ability to take several forms. By supporting polymorphism, object-oriented systems can send the same message to a set of objects, which can be

[2] For example, OPEN [I. Graham, B. Henderson-Seller, and H. Yanoussi, *The Open Process Specification* (Reading, MA: Addison-Wesley, 1997)], RUP [P. Kruchten, *The Rational Unified Process: An Introduction*, 2nd ed. (Reading, MA: Addison-Wesley, 2000)], and the Enhanced Unified Process (see Chapter 1).

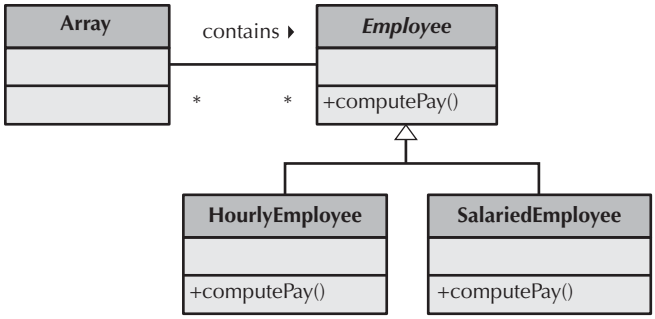


FIGURE 8-2
Example of Polymorphism

interpreted differently by different classes of objects. Based on encapsulation and information hiding, an object does not have to be concerned with *how* something is done when using other objects. It simply sends a message to an object and that object determines how to interpret the message. This is accomplished through the use of dynamic binding.

Dynamic binding refers to the ability of object-oriented systems to defer the data typing of objects to run time. For example, imagine that you have an array of type employee that contains instances of hourly employees and salaried employees (see Figure 8-2). Both these types of employees implement a compute pay method. An object can send the message to each instance contained in the array to compute the pay for that individual instance. Depending on whether the instance is an hourly employee or a salaried employee, a different method will be executed. The specific method is chosen at run time. With this ability, individual classes are easier to understand. However, the specific level of support for polymorphism and dynamic binding is language specific. Most object-oriented programming languages support dynamic binding of methods, and some support dynamic binding of attributes.

But polymorphism can be a double-edged sword. Through the use of dynamic binding, there is no way to know before run time which specific object will be asked to execute its method. In effect, there is a decision made by the system that is not coded anywhere.[3] Because all these decisions are made at run time, it is possible to send a message to an object that it does not understand (i.e., the object does not have a corresponding method). This can cause a run-time error that, if the system is not programmed to handle it correctly, can cause the system to abort.[4]

Finally, if the methods are not semantically consistent, the developer cannot assume that all methods with the same name will perform the same generic operation. For example, imagine that you have an array of type person that contains instances of employees and customers (see Figure 8-3). These both implement a compute pay method. An object can send the message to each instance contained in the array to execute the compute pay method for that individual instance. In the case of an instance of employee, the compute pay method computes the amount that the employee is owed by the firm, whereas the compute pay method associated with an instance of a customer computes the amount owed the firm by the customer. Depending on whether the instance is an employee or a customer, a different meaning is associated with the method. Therefore, the semantics of each method must be determined individually. This substantially increases the difficulty of understanding individual objects. The key to controlling the difficulty of understanding object-oriented systems

[3] From a practical perspective, there is an implied case statement. The system chooses the method based on the type of object being asked to execute it and the parameters passed as arguments to the method. This is typically done through message dispatch tables that are hidden from the programmer.

[4] In most object-oriented programming languages, these errors are referred to as exceptions that the system "throws" and must "catch." In other words, the programmer must correctly program the throw and catch or the systems will abort. Again, each programming language can handle these situations in a unique manner.

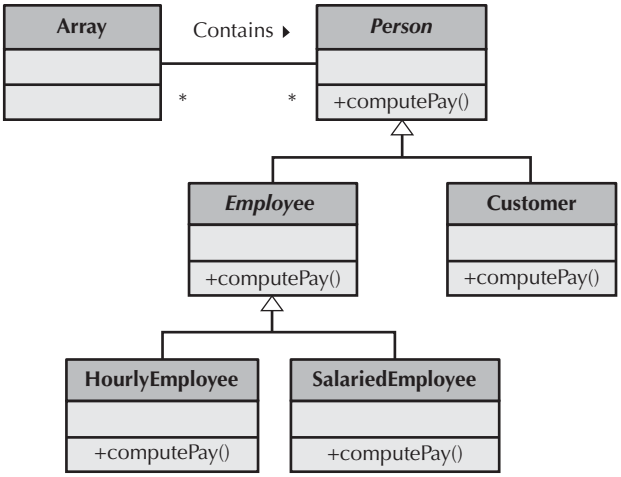


FIGURE 8-3
Example of Polymorphism Misuse

when using polymorphism is to ensure that all methods with the same name implement that same generic operation (i.e., they are semantically consistent).

Inheritance

Inheritance allows developers to define classes incrementally by reusing classes defined previously as the basis for new classes. Although we could define each class separately, it might be simpler to define one general superclass that contains the data and methods needed by the subclasses and then have these classes inherit the properties of the superclass. Subclasses inherit the attributes and methods from the superclasses above them. Inheritance makes it simpler to define classes.

There have been many different types of inheritance mechanisms associated with object-oriented systems.[5] The most common inheritance mechanisms include different forms of single and multiple inheritance. *Single inheritance* allows a subclass to have only a single parent class. Currently, all object-oriented methodologies, databases, and programming languages permit extending the definition of the superclass through single inheritance.

Some object-oriented methodologies, databases, and programming languages allow a subclass to redefine some or all the attributes and/or methods of its superclass. With *redefinition* capabilities, it is possible to introduce an *inheritance conflict* [i.e., an attribute (or method) of a subclass with the same name as an attribute (or method) of a super-class]. For example in Figure 8-4, Doctor is a subclass of Employee. Both have methods named ComputePay(). This causes an inheritance conflict. Furthermore, when the definition of a superclass is modified, all its subclasses are affected. This can introduce additional inheritance conflicts in one (or more) of the superclass's subclasses. For example in Figure 8-4, Employee could be modified to include an additional method, UpdateSchedule(). This would add another inheritance conflict between Employee and Doctor. Therefore, developers must be aware of the effects of the modification not only in the superclass but also in each subclass that inherits the modification.

Finally, through redefinition capabilities, it is possible for a programmer to arbitrarily cancel the inheritance of methods by placing stubs[6] in the subclass that will override the

[5] See, for example, M. Lenzerini, D. Nardi, and M. Simi, *Inheritance Hierarchies in Knowledge Representation and Programming Languages* (New York: Wiley, 1991).

[6] In this case, a stub is simply the minimal definition of a method to prevent syntax errors from occurring.

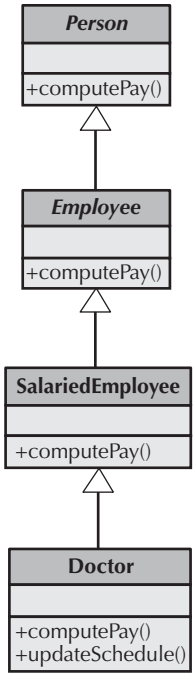


FIGURE 8-4
Example of Redefinition and Inheritance Conflict

definition of the inherited method. If the cancellation of methods is necessary for the correct definition of the subclass, then it is likely that the subclass has been misclassified (i.e., it is inheriting from the wrong superclass).

As you can see, from a design perspective, inheritance conflicts and redefinition can cause all kinds of problems with interpreting the final design and implementation.[7] However, most inheritance conflicts are due to poor classification of the subclass in the inheritance hierarchy (the generalization a-kind-of semantics are violated), or the actual inheritance mechanism violates the encapsulation and information hiding principle (i.e., subclasses are capable of directly addressing the attributes or methods of a superclass). To address these issues, Jim Rumbaugh and his colleagues suggested the following guidelines:[8]

- Do not redefine query operations.
- Methods that redefine inherited ones should restrict only the semantics of the inherited ones.
- The underlying semantics of the inherited method should never be changed.
- The signature (argument list) of the inherited method should never be changed.

However, many existing object-oriented programming languages violate these guidelines. When it comes to implementing the design, different object-oriented programming languages address inheritance conflicts differently. Therefore, it is important at this point in the development of the system to know what the chosen programming language supports. We must be sure that the design can be implemented as intended. Otherwise, the design needs to be modified before it is turned over to remotely located programmers.

When considering the interaction of inheritance with polymorphism and dynamic binding, object-oriented systems provide the developer with a very powerful, but dangerous, set of tools. Depending on the object-oriented programming language used, this interaction can allow the same object to be associated with different classes at different times. For example, an instance of Doctor can be treated as an instance of Employee or any of its direct and indirect superclasses, such as SalariedEmployee and Person, respectively (see Figure 8-4). Therefore, depending on whether static or dynamic binding is supported, the same object may execute different implementations of the same method at different times. Or, if the method is defined only with the SalariedEmployee class and it is currently treated as an instance of the Employee class, the instance could cause a run-time error to occur.[9] It is important to know what object-oriented programming language is going to be used so that these kinds of issues can be solved with the design, instead of the implementation, of the class.

With *multiple inheritance*, a subclass may inherit from more than one superclass. In this situation, the types of inheritance conflicts are multiplied. In addition to the possibility of having an inheritance conflict between the subclass and one (or more) of its superclasses, it is now possible to have conflicts between two (or more) superclasses. In this latter case, three different types of additional inheritance conflicts can occur:

[7] For more information, see Ronald J. Brachman, "I Lied about the Trees Or, Defaults and Definitions in Knowledge Representation," *AI Magazine* 5, no. 3 (Fall 1985): 80–93.

[8] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, *Object-Oriented Modeling and Design* (Englewood Cliffs, NJ: Prentice Hall, 1991).

[9] This happens with novices quite regularly when using C++.

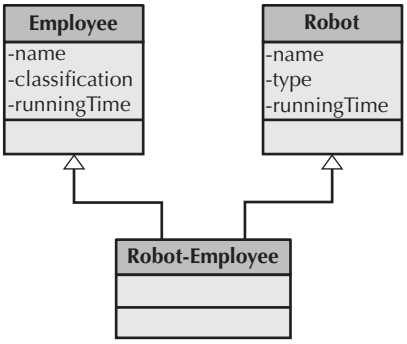


FIGURE 8-5
Additional Inheritance Conflicts with multiple Inheritance

- Two inherited attributes (or methods) have the same name (spelling) and semantics.
- Two inherited attributes (or methods) have different names but identical semantics (i.e., they are *synonyms*).
- Two inherited attributes (or methods) have the same name but different semantics (i.e., they are *heteronyms, homographs,* or *homonyms*). This also violates the proper use of polymorphism.

For example, in Figure 8-5, Robot-Employee is a subclass of both Employee and Robot. In this case, Employee and Robot conflict with the attribute name. Which one should Robot-Employee inherit? Because they are the same, semantically speaking, does it really matter? It is also possible that Employee and Robot could have a semantic conflict on the classification and type attributes if they have the same semantics. Practically speaking, the only way to prevent this situation is for the developer to catch it during the design of the subclass. Finally, what if the runningTime attributes have different semantics? In the case of Employee objects, the runningTime attribute stores the employee's time running a mile, whereas the runningTime attribute for Robot objects stores the average time between check-ups. Should Robot-Employee inherit both of them? It really depends on whether the robot employees can run the mile or not. With the potential for these additional types of conflicts, there is a risk of decreasing the understandability in an object-oriented system instead of increasing it through the use of multiple inheritance. Our advice is to use great care when using multiple inheritance.

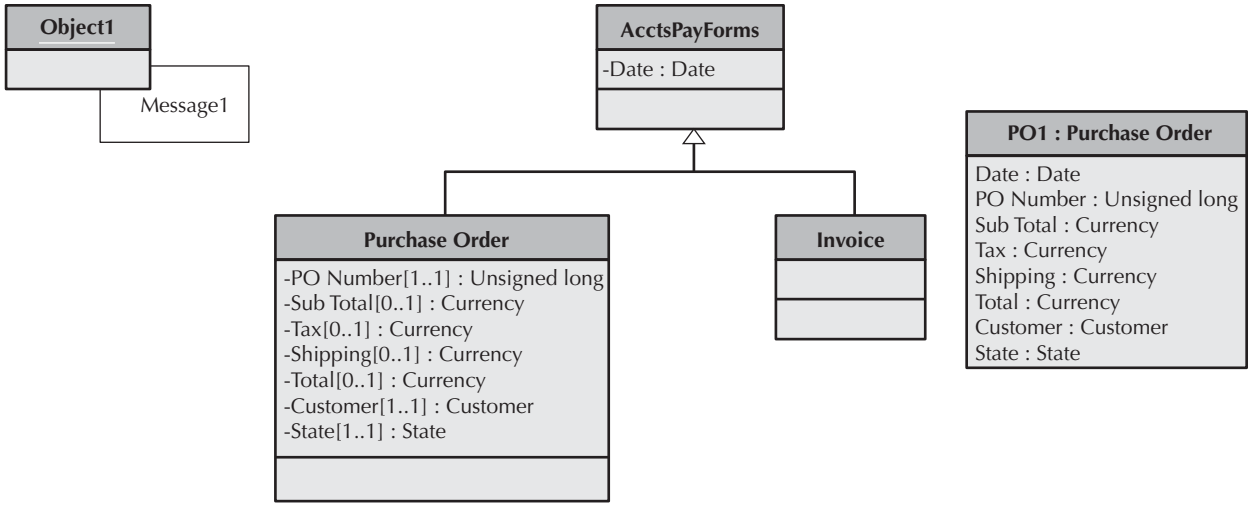
DESIGN CRITERIA

When considering the design of an object-oriented system, a set of criteria exists that can be used to determine whether the design is a good one or a bad one. According to Coad and Yourdon,[10] "A good design is one that balances trade-offs to minimize the total cost of the system over its entire lifetime." These criteria include coupling, cohesion, and connascence.

Coupling

Coupling refers to how interdependent or interrelated the modules (classes, objects, and methods) are in a system. The higher the interdependency, the more likely changes in part of a design

[10] Peter Coad and Edward Yourdon, *Object-Oriented Design* (Englewood Cliffs, NJ: Yourdon Press, 1991), p. 128.



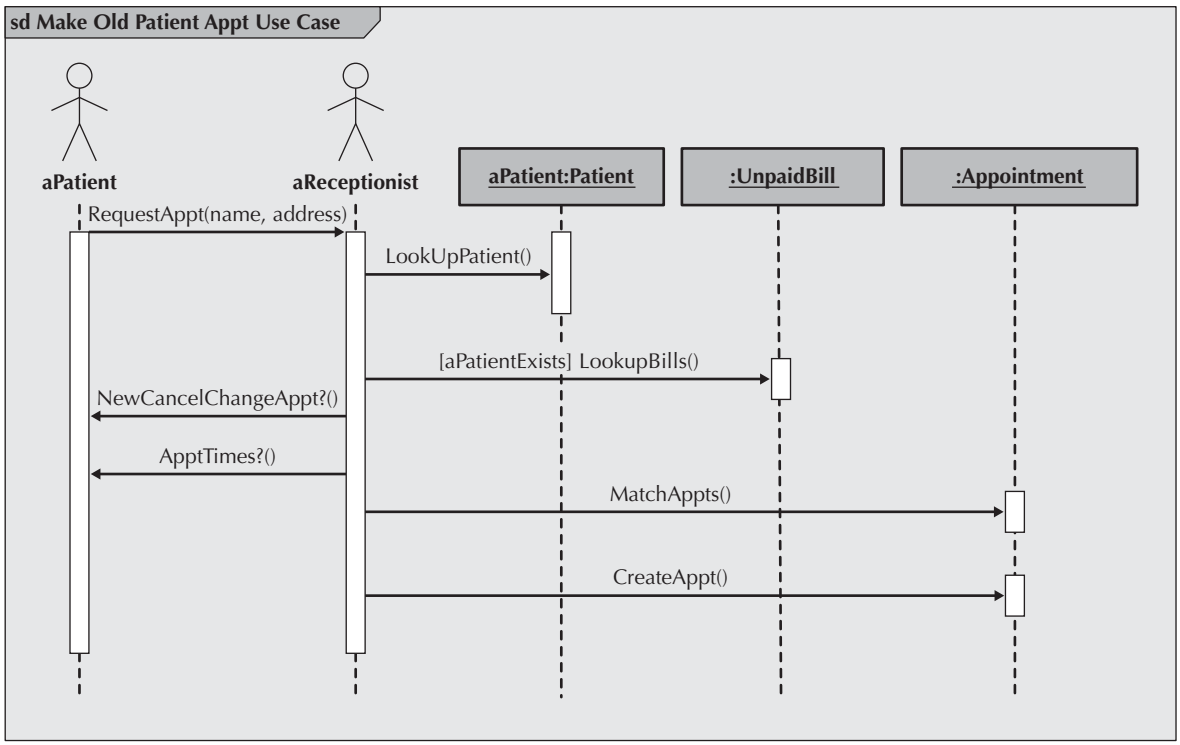


FIGURE 8-6 Examples of Interaction Coupling

can cause changes to be required in other parts of the design. For object-oriented systems, Coad and Yourdon[11] identified two types of coupling to consider: interaction and inheritance.

Interaction coupling deals with the coupling among methods and objects through message passing. Lieberherr and Holland put forth the *law of Demeter* as a guideline to minimize this type

[11] Ibid.

of coupling.[12] Essentially, the law minimizes the number of objects that can receive messages from a given object. The law states that an object should send messages only to one of the following:

- Itself (For example in Figure 8-6a, Object1 can send Message1 to itself. In other words, a method associated with Object1 can use other methods associated with Object1.[13])
- An object that is contained in an attribute of the object or one of its superclasses (For example in Figure 8-6b, the P01 instance of the Purchase Order class should be able to send messages using its Customer, State, and Date attributes.)
- An object that is passed as a parameter to the method (For example in Figure 8-6c, the aPatient instance sends the message RequestAppt(name, address) to the aReceptionist instance, which is allowed to send messages to the instances contained in the name and address parameters.)
- An object that is created by the method (For example in Figure 8-6c, the method RequestAppt associated with the aReceptionist instance creates an instance of the Appointment class. The RequestAppt method is allowed to send messages to that instance.)
- An object that is stored in a global variable[14]

Even though the law of Demeter attempts to minimize interaction coupling among methods and objects, each of the above allowed forms of message sending in fact increases coupling. For example, the coupling increases between the objects if the calling method passes attributes to the called method or if the calling method depends on the value being returned by the called method.

There are six types of interaction coupling, each falling on different parts of a good-to-bad continuum. They range from no direct coupling to content coupling. Figure 8-7 presents the

Level	Type	Description
Good	No Direct Coupling	The methods do not relate to one another; that is, they do not call one another.
	Data	The calling method passes a variable to the called method. If the variable is composite (i.e., an object), the entire object is used by the called method to perform its function.
	Stamp	The calling method passes a composite variable (i.e., an object) to the called method, but the called method only uses a portion of the object to perform its function.
	Control	The calling method passes a control variable whose value will control the execution of the called method.
	Common or Global	The methods refer to a "global data area" that is outside the individual objects.
Bad	Content or Pathological	A method of one object refers to the inside (hidden parts) of another object. This violates the principles of encapsulation and information hiding. However, C++ allows this to take place through the use of "friends."

Source: These types are based on material from Meilir Page-Jones, *The Practical Guide to Structured Systems Design,* 2nd Ed. (Englewood Cliffs, NJ: Yardon Press, 1988); Glenford Myers, *Composite/ Structured Design* (New York: Van Nostrand Reinhold, 1978).

FIGURE 8-7
Types of Interaction Coupling

[12] Karl J. Lieberherr and Ian M. Holland, "Assuring Good Style for Object-Oriented Programs," *IEEE Software* 6, no. 5 (September, 1989): 38–48; Karl J. Lieberherr, *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns* (Boston, MA: PWS Publishing, 1996).

[13] Obviously, this is stating what is expected.

[14] From a design perspective, global variables should be avoided. Most pure object-oriented programming languages do not explicitly support global variables, and we do not address them any further.

different types of interaction coupling. In general, interaction coupling should be minimized. The one possible exception is that non–problem-domain classes must be coupled to their corresponding problem-domain classes. For example, a report object (on the human–computer interaction layer) that displays the contents of an employee object (on the problem-domain layer) will be dependent on the employee object. In this case, for optimization purposes, the report class may be even content or pathologically coupled to the employee class. However, problem-domain classes should never be coupled to non–problem-domain classes.

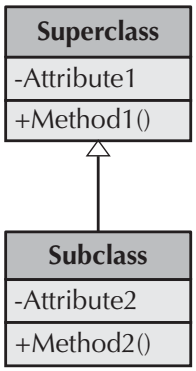


FIGURE 8-8
Example of Inheritance Coupling

Inheritance coupling, as its name implies, deals with how tightly coupled the classes are in an inheritance hierarchy. Most authors tend to say simply that this type of coupling is desirable. However, depending on the issues raised previously with inheritance—inheritance conflicts, redefinition capabilities, and dynamic binding—a high level of inheritance coupling might not be a good thing. For example, in Figure 8-8, should Method2() defined in Subclass be allowed to call Method1() defined in Superclass? Or, should Method2() defined in Subclass refer to Attribute1 defined in Superclass? Or, even more confusing, assuming that Superclass is an abstract class, can a Method1() call Method2() or use Attribute2 defined in Subclass? Obviously, the first two examples have some intuitive sense. Using the properties of a superclass is the primary purpose of inheriting from it in the first place. On the other hand, the third example is somewhat counterintuitive. However, owing to the way that different object-oriented programming languages support dynamic binding, polymorphism, and inheritance, all these examples could be possible.

As Snyder has pointed out, most problems with inheritance involve the ability within the object-oriented programming languages to violate the encapsulation and information-hiding principles.[15] From a design perspective, the developer needs to optimize the trade-offs of violating the encapsulation and information-hiding principles and increasing the desirable coupling between subclasses and its superclasses. The best way to solve this conundrum is to ensure that inheritance is used only to support generalization/specialization (a-kind-of) semantics and the principle of substitutability (see Chapter 5). All other uses should be avoided.

Cohesion

Cohesion refers to how single-minded a module (class, object, or method) is within a system. A class or object should represent only one thing, and a method should solve only a single task. Three general types of cohesion have been identified by Coad and Yourdon for object-oriented systems: method, class, and generalization/specialization.[16]

Method cohesion addresses the cohesion within an individual method (i.e., how single-minded a method is). Methods should do one and only one thing. A method that actually performs multiple functions is more difficult to understand—and, therefore, to implement and maintain—than one that performs only a single function. Seven types of method cohesion have been identified (see Figure 8-9). They range from functional

[15] Alan Snyder, "Encapsulation and Inheritance in Object-Oriented Programming Languages," in N. Meyrowitz (ed.), *OOPSLA '86 Conference Proceedings, ACM SigPlan Notices* 21, no. 11 (November 1986); Alan Snyder, "Inheritance and the Development of Encapsulated Software Components," in B. Shriver and P. Wegner (eds.), *Research Directions in Object-Oriented Programming* (Cambridge, MA: MIT Press, 1987).

[16] Coad and Yourdon, *Object-Oriented Design*.

cohesion (good) down to coincidental cohesion (bad). In general, method cohesion should be maximized.

Class cohesion is the level of cohesion among the attributes and methods of a class (i.e., how single-minded a class is). A class should represent only one thing, such as an employee, a department, or an order. All attributes and methods contained in a class should be required for the class to represent the thing. For example, an employee class should have attributes that deal with a social security number, last name, first name, middle initial, addresses, and benefits, but it should not have attributes such as door, engine, or hood. Furthermore, there should be no attributes or methods that are never used. In other words, a class should have only the attributes and methods necessary to fully define instances for the problem at hand. In this case, we have *ideal class cohesion*. Glenford Meyers suggested that a cohesive class[17] should have these attributes:

- It should contain multiple methods that are visible outside the class (i.e., a single-method class rarely makes sense).
- Each visible method performs only a single function (i.e., it has functional cohesion; see Figure 8-9).

Level	Type	Description
Good	Functional	A method performs a single problem-related task (e.g., calculate current GPA).
	Sequential	The method combines two functions in which the output from the first one is used as the input to the second one (e.g., format and validate current GPA).
	Communicational	The method combines two functions that use the same attributes to execute (e.g., calculate current and cumulative GPA).
	Procedural	The method supports multiple weakly related functions. For example, the method could calculate student GPA, print student record, calculate cumulative GPA, and print cumulative GPA.
	Temporal or Classical	The method supports multiple related functions in time (e.g., initialize all attributes).
	Logical	The method supports multiple related functions, but the choice of the specific function is chosen based on a control variable that is passed into the method. For example, the called method could open a checking account, open a savings account, or calculate a loan, depending on the message that is sent by its calling method.
Bad	Coincidental	The purpose of the method cannot be defined or it performs multiple functions that are unrelated to one another. For example, the method could update customer records, calculate loan payments, print exception reports, and analyze competitor pricing structure.

Source: These types are based on material from Page-Jones, *The Practical Guide to Structured Systems*; Myers, *Composite/Structured Design*; Edward Yourdon and Larry L. Constantine, *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design* (Englewood Cliffs, NJ: Prentice-Hall, 1979).

FIGURE 8-9
Types of Method Cohesion

[17] We have adapted his informational-strength module criteria from structured design to object-oriented design. [See Glenford J. Myers, *Composite/Structured Design* (New York, NY: Van Nostrand Reinhold, 1978).]

- All methods reference only attributes or other methods defined within the class or one of its superclasses (i.e., if a method is going to send a message to another object, the remote object must be the value of one of the local object's attributes).[18]
- It should not have any control couplings between its visible methods (see Figure 8-7).

Page-Jones[19] has identified three less-than-desirable types of class cohesion: mixed-instance, mixed-domain, and mixed-role (see Figure 8-10). An individual class can have a mixture of any of the three types.

Generalization/specialization cohesion addresses the sensibility of the inheritance hierarchy. How are the classes in the inheritance hierarchy related? Are the classes related through a generalization/specialization (a-kind-of) semantics? Or, are they related via some association, aggregation, or membership type of relationship that was created for simple reuse purposes? Recall all the issues raised previously on the use of inheritance. For example, in Figure 8-11, the subclasses ClassRooms and Staff inherit from the superclass Department. Obviously, instances of the ClassRooms and Staff classes are not a-kind-of Department. However, in the early days of object-oriented programming, this use of inheritance was quite common. When a programmer saw that there were some common properties that a set of classes shared, the programmer would create an artificial abstraction that defined the commonalities. This was potentially useful in a reuse sense, but it turned out to cause many maintenance nightmares. In this case, instances of the ClassRooms and Staff classes are associated with or a-part-of an instance of Department. Today we know that highly cohesive inheritance hierarchies should support only the semantics of generalization and specialization (a-kind-of) and the principle of substitutability.

Level	Type	Description
Good	Ideal	The class has none of the mixed cohesions.
↓	Mixed-Role	The class has one or more attributes that relate objects of the class to other objects on the same layer (e.g., the problem domain layer), but the attribute(s) has nothing to do with the underlying semantics of the class.
↓	Mixed-Domain	The class has one or more attributes that relate objects of the class to other objects on a different layer. As such, they have nothing to do with the underlying semantics of the thing that the class represents. In these cases, the offending attribute(s) belongs in another class located on one of the other layers. For example, a port attribute located in a problem domain class should be in a system architecture class that is related to the problem domain class.
Worse	Mixed-Instance	The class represents two different types of objects. The class should be decomposed into two separate classes. Typically, different instances only use a portion of the full definition of the class.

Based upon material from Page-Jones, *Fundamentals of Object-Oriented Design in UML*.

FIGURE 8-10
Types of Class Cohesion

[18] This restricts messages passing to only the first, second, and fourth conditions supported by the law of Demeter. For example, in Figure 8-6c, aReceptionist must have attributes associated with it that contains objects for Patients, Unpaid Bills, and Appointments. Furthermore, once an instance of Appointment is created, aReceptionist must have an attribute with the instance as its value to send any additional messages.

[19] See Meilir Page-Jones, *Fundamentals of Object-Oriented Design in UML* (Reading, MA: Addison-Wesley, 2000).

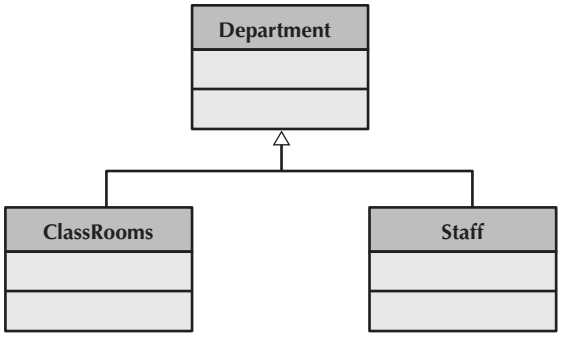


FIGURE 8-11
Generalization/ Specialization vs. Inheritance Abuse

Connascence

Connascence[20] generalizes the ideas of cohesion and coupling, and it combines them with the arguments for encapsulation. To accomplish this, three levels of encapsulation have been identified. Level-0 encapsulation refers to the amount of encapsulation realized in an individual line of code, level-1 encapsulation is the level of encapsulation attained by combining lines of code into a method, and level-2 encapsulation is achieved by creating classes that contain both methods and attributes. Method cohesion and interaction coupling address primarily level-1 encapsulation. Class cohesion, generalization/specialization cohesion, and inheritance coupling address only level-2 encapsulation. Connascence, as a generalization of cohesion and coupling, addresses both level-1 and level-2 encapsulation.

But what exactly is connascence? Connascence literally means to be born together. From an object-oriented design perspective, it really means that two modules (classes or methods) are so intertwined that if you make a change in one, it is likely that a change in the other will be required. On the surface, this is very similar to coupling and, as such, should be minimized. However, when you combine it with the encapsulation levels, it is not quite that simple. In this case, we want to minimize overall connascence by eliminating any unnecessary connascence throughout the system; minimize connascence across any encapsulation boundaries, such as method boundaries and class boundaries; and maximize connascence within any encapsulation boundary.

Based on these guidelines, a subclass should never directly access any hidden attribute or method of a superclass [i.e., a subclass should not have special rights to the properties of its superclass(es)]. If direct access to the nonvisible attributes and methods of a superclass by its subclass is allowed—and is permitted in most object-oriented programming languages—and a modification to the superclass is made, then owing to the connascence between the subclass and its superclass, it is likely that a modification to the subclass also is required.[21] In other words, the subclass has access to something across an encapsulation boundary (the class boundary between the subclass and the superclass). Practically speaking, you should maximize the cohesion (connascence) within an encapsulation boundary and minimize the coupling (connascence) between the encapsulation boundaries. There are many possible types of connascence. Figure 8-12 describes five of the types.

[20] See Meilir Page-Jones, "Comparing Techniques by Means of Encapsulation and Connascence," *Communications of the ACM* 35, no. 9 (September 1992): 147–151.

[21] Based on these guidelines, the use of the protected visibility, as supported in Java and C++, should be minimized, if not avoided. "Friends" as defined in C++ also should be minimized or avoided. Owing to the level of dependencies these language features create, any convenience afforded to a programmer is more than offset in potential design, understandability, and maintenance problems. These features must be used with great caution and must be fully documented.

Type	Description
Name	If a method refers to an attribute, it is tied to the name of the attribute. If the attribute's name changes, the content of the method will have to change.
Type or Class	If a class has an attribute of type A, it is tied to the type of the attribute. If the type of the attribute changes, the attribute declaration will have to change.
Convention	A class has an attribute in which a range of values has a semantic meaning (e.g., account numbers whose values range from 1000 to 1999 are assets). If the range would change, then every method that used the attribute would have to be modified.
Algorithm	Two different methods of a class are dependent on the same algorithm to execute correctly (e.g., insert an element into an array and find an element in the same array). If the underlying algorithm would change, then the insert and find methods would also have to change.
Position	The order of the code in a method or the order of the arguments to a method is critical for the method to execute correctly. If either is wrong, then the method will, at least, not function correctly.

Based upon material from Meilir Page-Jones, "Comparing Techniques by Means of Encapsulation and Connascence" and Meilir Page-Jones, *Fundamentals of Object-Oriented Design in UML*.

FIGURE 8-12
Types of Connascence

OBJECT DESIGN ACTIVITIES

The design activities for classes and methods are really an extension of the analysis and evolution activities presented previously (see Chapters 4 through 7). In this case, we expand the descriptions of the partitions, layers, and classes. Practically speaking, the expanded descriptions are created through the activities that take place during the detailed design of the classes and methods. The activities used to design classes and methods include additional specification of the current model, identifying opportunities for reuse, restructuring the design, optimizing the design, and, finally, mapping the problem-domain classes to an implementation language. Of course, any changes made to a class on one layer can cause the classes on the other layers that are coupled to it to be modified as well.

Adding Specifications

At this point in the development of the system, it is crucial to review the current set of functional, structural, and behavioral models. First, we should ensure that the classes on the problem-domain layer are both necessary and sufficient to solve the underlying problem. To do this, we need to be sure that there are no missing attributes or methods and no extra or unused attributes or methods in each class. Furthermore, are there any missing or extra classes? If we have done our job well during analysis, there will be few, if any, attributes, methods, or classes to add to the models. And it is unlikely that we have any extra attributes, methods, or classes to delete from the models. However, we still need to ensure that we have factored, abstracted, and refined the evolving models and created the relevant partitions and collaborations (see Chapter 7).

Second, we need to finalize the visibility (hidden or visible) of the attributes and methods in each class. Depending on the object-oriented programming language used, this could be predetermined. [For example, in Smalltalk, attributes are hidden and methods are visible. Other languages allow the programmer to set the visibility of each attribute or method. For example, in C^{++} and Java, you can set the visibility to private (hidden), public (visible), or

protected (visible to subclasses, but not to other classes).][22] By default, most object-oriented analysis and design approaches assume Smalltalk's approach.

Third, we need to decide on the signature of every method in every class. The *signature* of a method comprises three parts: the name of the method, the parameters or arguments that must be passed to the method, including their object type, and the type of value that the method will return to the calling method. The signature of a method is related to the method's *contract.*[23]

Fourth, we need to define any constraints that must be preserved by the objects (e.g., an attribute of an object that can have values only in a certain range). There are three different types of constraints: preconditions, postconditions, and invariants.[24] These are captured in the form of contracts and assertions added to the CRC cards and class diagrams. We also must decide how to handle a violation of a constraint. Should the system simply abort? Should the system automatically undo the change that caused the violation? Should the system let the end user determine the approach to correct the violation? In other words, the designer must design the errors that the system is expected to handle. It is best not to leave these types of design decisions for the programmer to solve. Violations of a constraint are known as *exceptions* in languages such as C++ and Java.

Even though we have described these activities in the context of the problem-domain layer, they are also applicable to the other layers: data management (Chapter 9), human–computer interaction (Chapter 10), and physical architecture (Chapter 11).

Identifying Opportunities for Reuse

Previously, we looked at possibly employing reuse in our models in analysis through the use of *patterns* (see Chapter 5). In design, in addition to using analysis patterns, there are opportunities for using design patterns, frameworks, libraries, and components. The opportunities vary depending on which layer is being reviewed. For example, it is doubtful that a class library will be of much help on the problem-domain layer, but a class library could be of great help on the foundation layer.

Like analysis patterns, design patterns are simply useful grouping of collaborating classes that provide a solution to a commonly occurring problem. The primary difference between analysis and design patterns is that design patterns are useful in solving "a general design problem in a particular context,"[25] whereas analysis patterns tended to aid in filling out a problem-domain representation. For example, a useful design pattern is the Whole-Part pattern (see Figure 8-13a). The Whole-Part pattern explicitly supports the Aggregation and Composition relationships within the UML. Another useful design pattern is the Iterator pattern (see Figure 8-13b). The primary purpose of the Iterator pattern is to provide the designer with a standard approach to support traversing different types of collections. By using this pattern, regardless of the collection type (ConcreteAggregate), the designer knows that the collection will need to create an iterator (ConcreteIterator) that customizes the standard operations used to traverse the collection: first(), next(), isDone(), and currentItem(). Given the number of collections typically found in business applications, this pattern is one of the more useful ones. For example in Figure 8-14a, we replicate a portion of both the Appointment and Library problems discussed in previous chapters, and in Figure 8-14b we show how the Iterator pattern can be

[22] It is also possible to control visibility through packages and friends (see Footnote 21).

[23] Contracts were introduced in Chapter 5, and they are described in detail later in this chapter.

[24] Constraints are described in more detail later in this chapter.

[25] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software* (Reading, MA: Addison-Wesley, 1995).

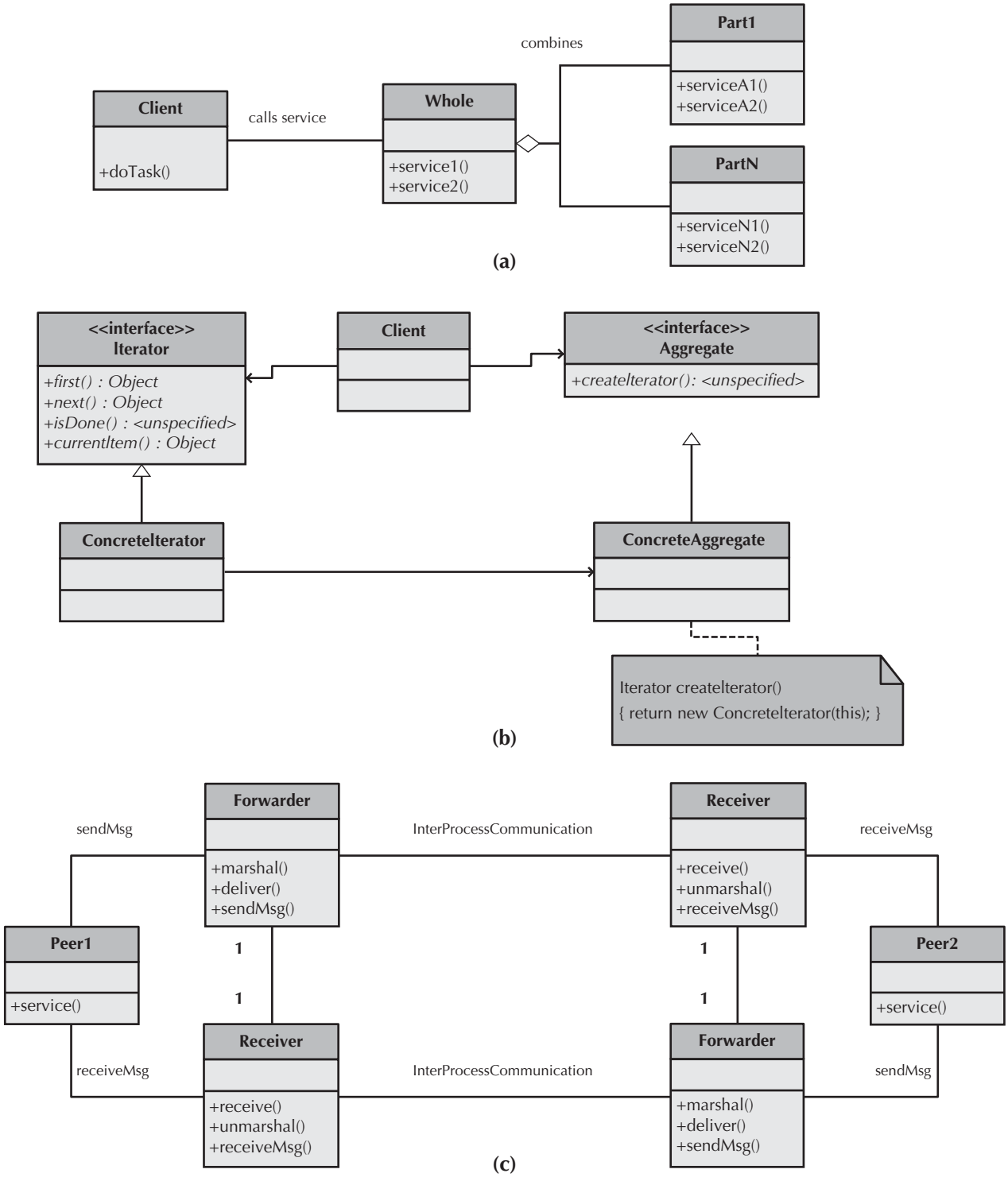


FIGURE 8-13 Sample Design Patterns

Source: Based upon material from F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture: A System of Patterns* (Chichester, UK: Wiley, 1996); E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software* (Reading, MA: Addison-Wesley, 1995).

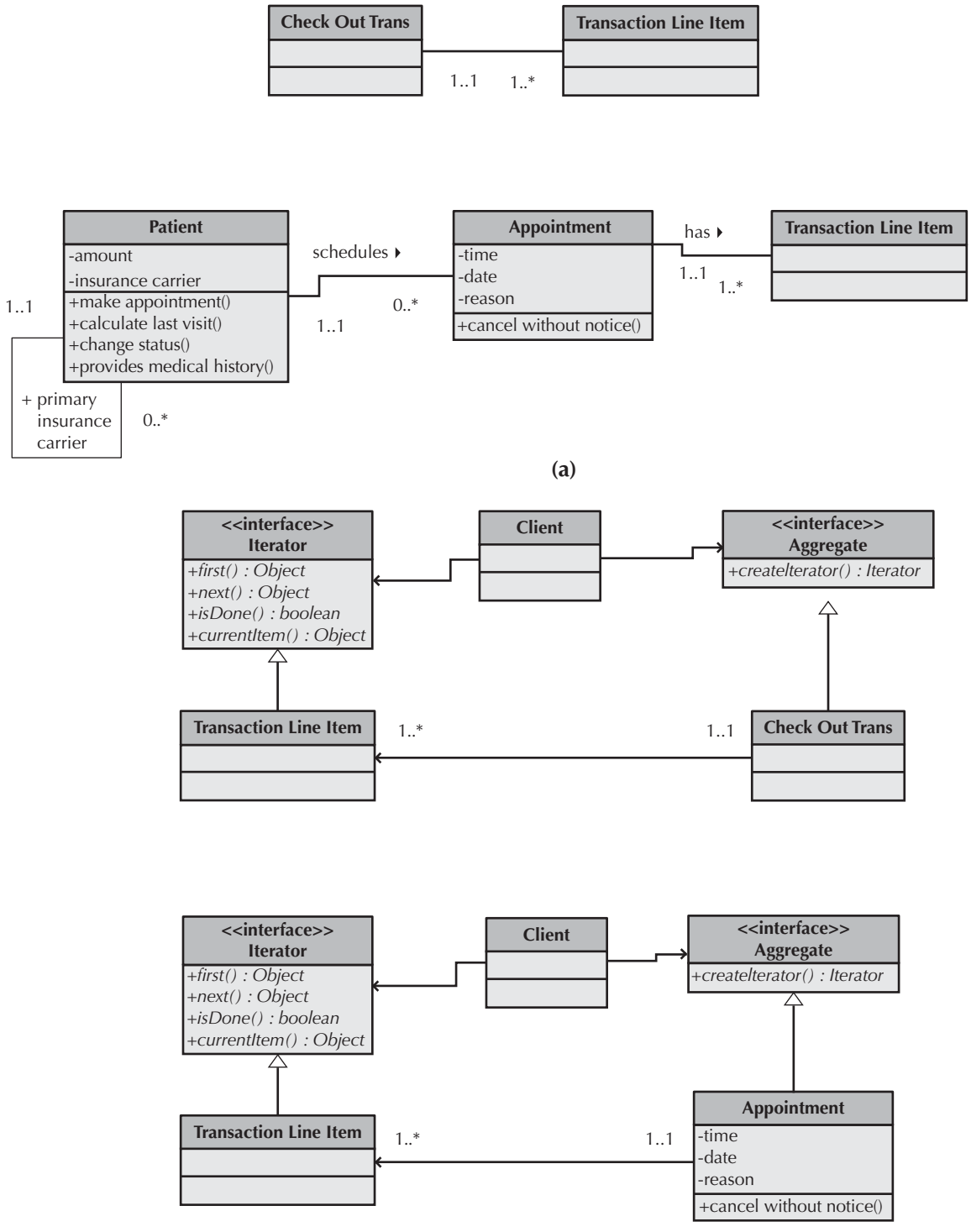


FIGURE 8-14 Iterator Design Pattern Applied to Library and Appointment Problems

applied to those sections of their evolving designs. Finally, some of the design patterns support different physical architectures (see Chapter 11). For example, the Forwarder-Receiver pattern (see Figure 8-13c) supports a peer-to-peer architecture. Many design patterns are available in C++ or Java source code.

A *framework* is composed of a set of implemented classes that can be used as a basis for implementing an application. Most frameworks allow us to create subclasses to inherit from classes in the framework. There are object-persistence frameworks that can be purchased and used to add persistence to the problem-domain classes, which would be helpful on the data management layer. Of course, when inheriting from classes in a framework, we are creating a dependency (i.e., increasing the inheritance coupling from the subclass to the superclass). Therefore, if we use a framework and the vendor makes changes to the framework, we will have to at least recompile the system when we upgrade to the new version of the framework.

A *class library* is similar to a framework in that it typically has a set of implemented classes that were designed for reuse. However, frameworks tend to be more domain specific. In fact, frameworks may be built using a class library. A typical class library could be purchased to support numerical or statistical processing, file management (data management layer), or user interface development (human–computer interaction layer). In some cases, instances of classes contained in the class library can be created, and in other cases, classes in the class library can be extended by creating subclasses based on them. As with frameworks, if we use inheritance to reuse the classes in the class library, we will run into all the issues dealing with inheritance coupling and connascence. If we directly instantiate classes in the class library, we will create a dependency between our object and the library object based on the signatures of the methods in the library object. This increases the interaction coupling between the class library object and our object.

A *component* is a self-contained, encapsulated piece of software that can be plugged into a system to provide a specific set of required functionalities. Today, there are many components available for purchase. A component has a well-defined *API* (application program interface). An API is essentially a set of method interfaces to the objects contained in the component. The internal workings of the component are hidden behind the API. Components can be implemented using class libraries and frameworks. However, components also can be used to implement frameworks. Unless the API changes between versions of the component, upgrading to a new version normally requires only linking the component back into the application. As such, recompilation typically is not required.

Which of these approaches should we use? It depends on what we are trying to build. In general, frameworks are used mostly to aid in developing objects on the physical architecture, human–computer interaction, or data management layers; components are used primarily to simplify the development of objects on the problem-domain and human–computer interaction layers; and class libraries are used to develop frameworks and components and to support the foundation layer. Whichever of these reuse approaches you use, you must remember that reuse brings many potential benefits and possible problems. For example, the software has previously been verified and validated, which should reduce the amount of testing required for our system. However as stated before, if the software on which we are basing our system changes, then most likely, we will also have to change our system. Furthermore, if the software is from a third-party firm, we are creating a dependency from our firm (or our client's firm) to the third-party vendor. Consequently, we need to have some confidence that the vendor will be in business for a while.

Restructuring the Design

Once the individual classes and methods have been specified and the class libraries, frameworks, and components have been incorporated into the evolving design, we should use

factoring to restructure the design. *Factoring* (Chapter 7) is the process of separating out aspects of a method or class into a new method or class to simplify the overall design. For example, when reviewing a set of classes on a particular layer, we might discover that a subset of them shares a similar definition. In that case, it may be useful to factor out the similarities and create a new class. Based on the issues related to cohesion, coupling, and connascence, the new class may be related to the old classes via inheritance (generalization) or through an aggregation or association relationship.

Another process that is useful for restructuring the evolving design is *normalization*. Normalization is described in Chapter 9 in relation to relational databases. However, normalization can be useful at times to identify potential classes that are missing from the design. Also related to normalization is the requirement to implement the actual association and aggregation relationships as attributes. Virtually no object-oriented programming language differentiates between attributes and association and aggregation relationships. Therefore, all association and aggregation relationships must be converted to attributes in the classes. For example in Figure 8-15a, the Customer and State classes are associated with the Order class. Furthermore, the Product-Order association class is associated with both the Order and Product classes. One of the first things that must be done is to convert the Product Order Association class to a normal class. Notice the multiplicity values for the new associations between the Order and the Product Order classes and the Product Order and Product classes (see Figure 8-15b). Next, we need to convert all associations to attributes that represent the relationships between the affected classes. In this case, the Customer class must have an Orders attribute added to represent the set of orders that an instance of the Customer class may possess; the Order class must add attributes to reference instances of the Customer, State, and Product Order classes; the State class must have an attribute added to it to reference all of the instances of the Order class that is associated with that particular state; the new Product Order class must have attributes that allow an instance of the Product Order class to reference which instance of the Order class and which instance of the Product class is relevant to it; and, finally, the Product class must add an attribute that references the relevant instances of the Product Order class (see Figure 8-15c). As you can see, even in this very small example, many changes need to be made to ready the design for implementation.

Finally, all inheritance relationships should be challenged to ensure that they support only a generalization/specialization (a-kind-of) semantics. Otherwise, all the problems mentioned previously with inheritance coupling, class cohesion, and generalization/specialization cohesion will come to pass.

Optimizing the Design[26]

Up until now, we have focused our energy on developing an understandable design. With all the classes, patterns, collaborations, partitions, and layers designed and with all the class libraries, frameworks, and components included in the design, understandability has been our primary focus. However, increasing the understandability of a design typically creates an inefficient design. Conversely, focusing on efficiency issues will deliver a design that is more difficult to understand. A good practical design manages the inevitable trade-offs that must occur.[27]

[26] The material contained in this section is based on James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen, *Object-Oriented Modeling and Design* (Englewood Cliffs, NJ: Prentice Hall, 1991); Bernd Brugge and Allen H. Dutoit, *Object-Oriented Software Engineering: Conquering Complex and Changing Systems* (Englewood Cliffs, NJ: Prentice Hall, 2000).

[27] The optimizations described here are only suggestions. In all cases, the decision to implement one or more of these optimizations really depends on the problem domain of the system and the environment on which the system will reside, i.e., the data management layer (see Chapter 9), the human–computer interaction layer (see Chapter 10), and the physical architecture layer (see Chapter 11).

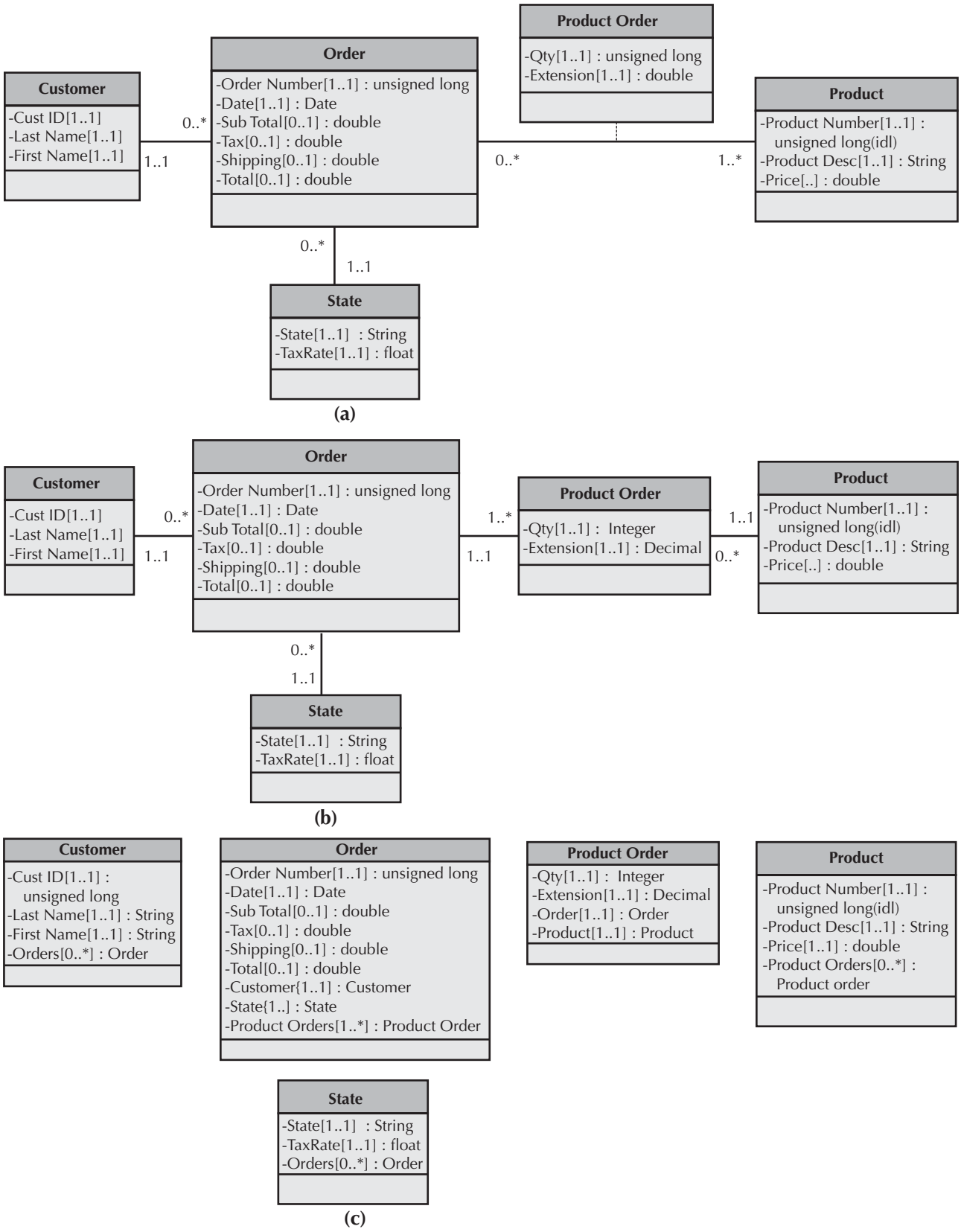


FIGURE 8-15 Converting Associations to Attributes

The first optimization to consider is to review the access paths between objects. In some cases, a message from one object to another has a long path to traverse (i.e., it goes through many objects). If the path is long and the message is sent frequently, a redundant path should be considered. Adding an attribute to the calling object that will store a direct connection to the object at the end of the path can accomplish this.

A second optimization is to review each attribute of each class. It should be determined which methods use the attributes and which objects use the methods. If the only methods that use an attribute are read and update methods and only instances of a single class send messages to read and update the attribute, then the attribute may belong with the calling class instead of the called class. Moving the attribute to the calling class will substantially speed up the system.

A third optimization is to review the direct and indirect fan-out of each method. *Fan-out* refers to the number of messages sent by a method. The direct fan-out is the number of messages sent by the method itself, whereas the indirect fan-out also includes the number of messages sent by the methods called by the other methods in a message tree. If the fan-out of a method is high relative to the other methods in the system, the method should be optimized. One way to do this is to consider adding an index to the attributes used to send the messages to the objects in the message tree.

A fourth optimization is to look at the execution order of the statements in often-used methods. In some cases, it is possible to rearrange some of the statements to be more efficient. For example, if based on the objects in the system, it is known that a search routine can be narrowed by searching on one attribute before another one, then the search algorithm should be optimized by forcing it to always search in a predefined order.

A fifth optimization is to avoid recomputation by creating a *derived attribute* (or *active value*) (e.g., a total that stores the value of the computation). This is also known as *caching computational results,* and it can be accomplished by adding a *trigger* to the attributes contained in the computation (i.e., attributes on which the derived attribute is dependent). This would require a recomputation to take place only when one of the attributes that go into the computation is changed. Another approach is to simply mark the derived attribute for recomputation and delay the recomputation until the next time the derived attribute is accessed. This last approach delays the recomputation as long as possible. In this manner, a computation does not occur unless it must occur. Otherwise, every time a derived attribute needs to be accessed, a computation will be required.

A sixth optimization that should be considered deals with objects that participate in a one-to-one association; that is, they both must exist for either to exist. In this case, it might make sense, for efficiency purposes, to collapse the two defining classes into a single class. However, this optimization might need to be reconsidered when storing the "fatter" object in a database. Depending on the type of object persistence used (see Chapter 9), it can actually be more efficient to keep the two classes separate. Alternatively, it could make more sense for the two classes to be combined on the problem-domain layer but kept separate on the data management layer.

Mapping Problem-Domain Classes to Implementation Languages[28]

Up until this point, it has been assumed that the classes and methods in the models would be implemented directly in an object-oriented programming language. However, now it is important to map the current design to the capabilities of the programming language used. For example, if we have used multiple inheritance in our design but we are implementing in a language that supports only single inheritance, then the multiple inheritance must be factored out of the design. If the implementation is to be done in an object-based language,

[28] The mapping rules presented in this section are based on material in Coad and Yourdon, *Object-Oriented Design*.

one that does not support inheritance,[29] or a non–object-based language, such as C, we must map the problem-domain objects to programming constructs that can be implemented using the chosen implementation environment.

Implementing Problem Domain Classes in a Single-Inheritance Language The only issue associated with implementing problem-domain objects is the factoring out of any multiple inheritance—i.e., the use of more than one superclass—used in the evolving design. For example, if you were to implement the solution in Java, Smalltalk, or Visual Basic.net, you must factor out any multiple inheritance. The easiest way to do this is to use the following rule:

> **RULE 1a:** Convert the additional inheritance relationships to association relationships. The multiplicity of the new association from the subclass to the superclass should be 1..1. If the additional superclasses are concrete, that is, they can be instantiated themselves, then the multiplicity from the superclass to the subclass is 0..1. Otherwise, it is 1..1. Furthermore, an exclusive-or (XOR) constraint must be added between the associations. Finally, you must add appropriate methods to ensure that all information is still available to the original class.
>
> ***or***
>
> **RULE 1b:** Flatten the inheritance hierarchy by copying the attributes and methods of the additional superclass(es) down to all of the subclasses and remove the additional superclass from the design.[30]

Figure 8-16 demonstrates the application of these rules. Figure 8-16a portrays a simple example of multiple inheritance where Flying Car inherits from both Airplane and Car, and Amphibious Car inherits from both Car and Boat. Assuming that Car is concrete, we apply Rule 1a to part a, and we end up with the diagram in part b, where we have added the association between Flying Car and Car and the association between Amphibious Car and Boat. The multiplicities have been added correctly, and the XOR constraint has been applied. If we apply Rule 1b to part a, we end up with the diagram in part c, where all the attributes of Car have been copied down into Flying Car and Amphibious Car. In this latter case, you might have to deal with the effects of inheritance conflicts (see earlier in the chapter).

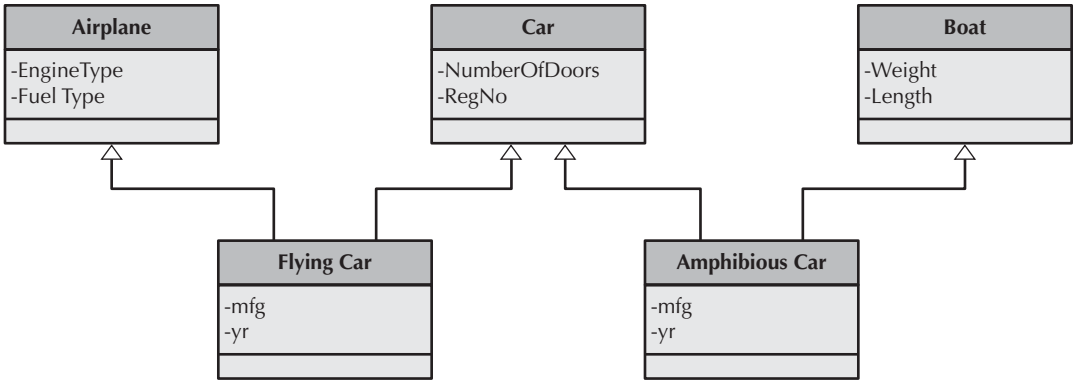
The advantage of Rule 1a is that all problem-domain classes identified during analysis are preserved. This allows maximum flexibility of maintenance of the design of the problem domain layer. However, Rule 1a increases the amount of message passing required in the system, and it has added processing requirements involving the XOR constraint, thus reducing the overall efficiency of the design. Accordingly, our recommendation is to limit Rule 1a to be applied only when dealing with "extra" superclasses that are concrete because they have an independent existence in the problem domain. Use Rule 1b when they are abstract because they do not have an independent existence from the subclass.

Implementing Problem Domain Objects in an Object-Based Language If we are going to implement our solution in an *object-based language* (i.e., a language that supports the creation of objects but does not support implementation inheritance), we must factor out all uses of inheritance from the problem-domain class design. Applying the preceding rule to all superclasses enables us to restructure our design without any inheritance.

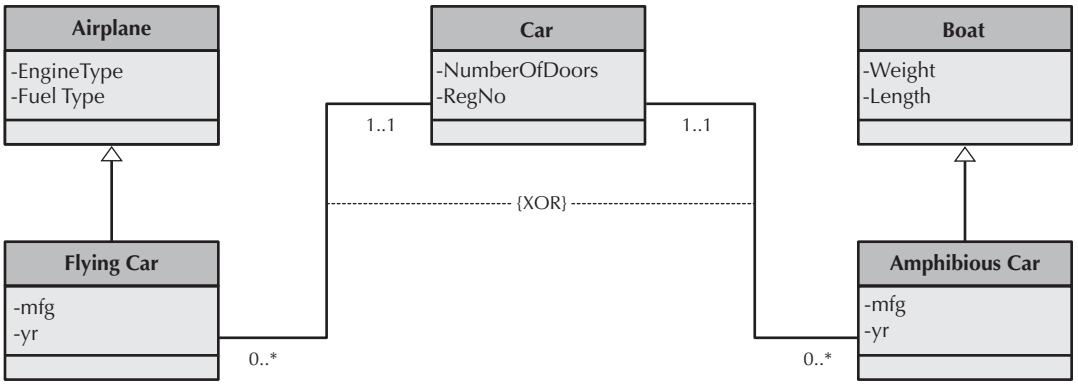
Figure 8-17 demonstrates the application of the preceding rules. Figure 8-17a shows the same simple example of multiple inheritance portrayed in Figure 8-16, where Flying Car

[29] In this case, we are talking about implementation inheritance, not the interface inheritance. Interface inheritance supported by Visual Basic and Java supports only inheriting the requirements to implement certain methods, not any implementation. Java and Visual Basic.net also support single inheritance as described in this text.

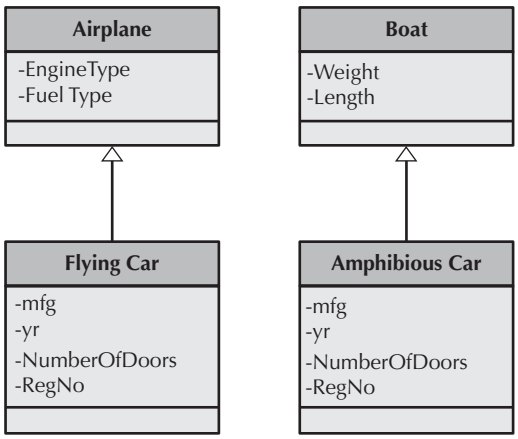
[30] It is also a good idea to document this modification in the design so that in the future, modifications to the design can be maintained easily.



(a)



(b)



(c)

FIGURE 8-16 Factoring Out Multiple-Inheritance Effect for a Single-Inheritance Language

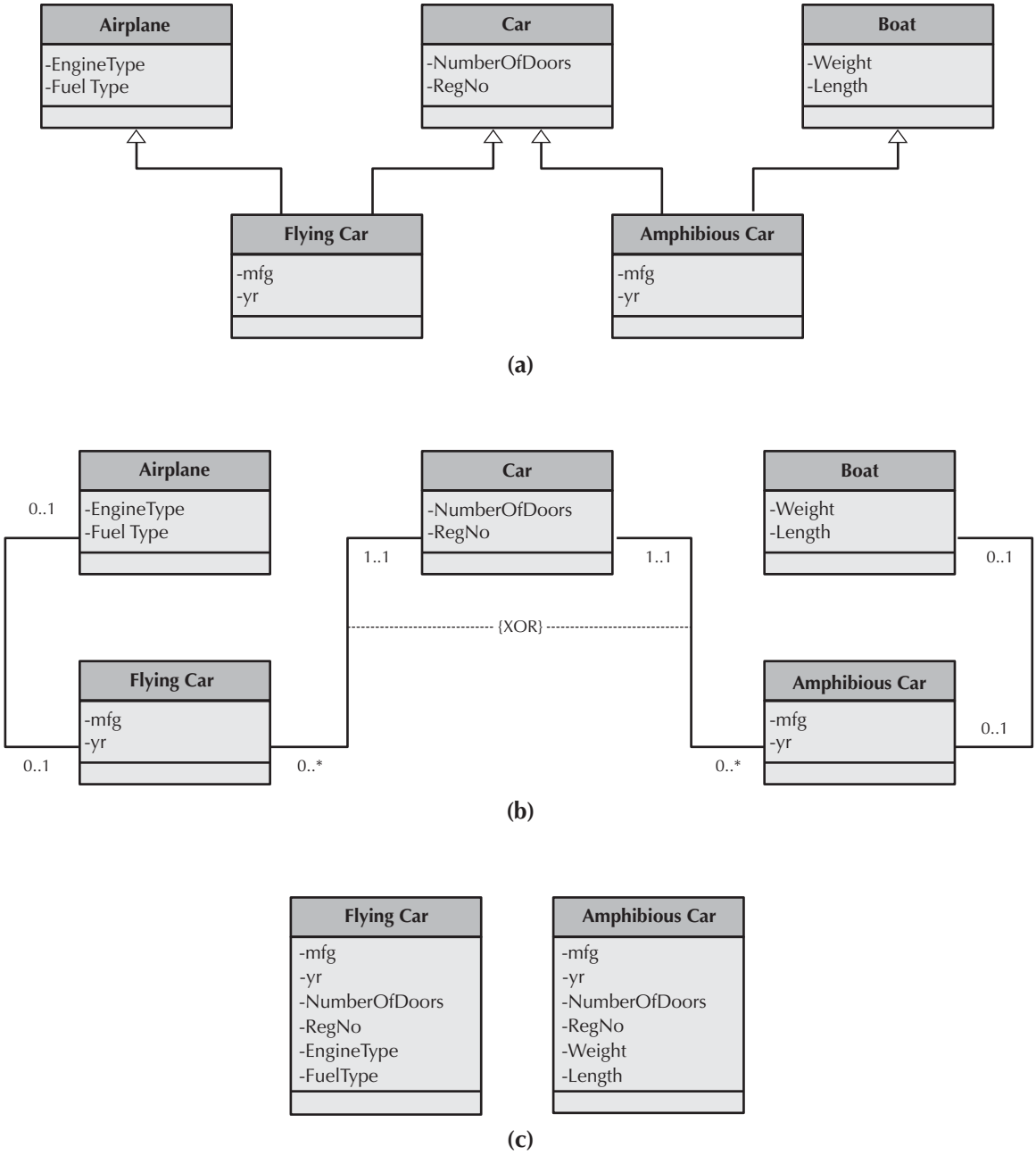


FIGURE 8-17 Factoring Out Multiple Inheritance Effect for an Object-Based Language

inherits from both Airplane and Car, and Amphibious Car inherits from both Car and Boat. Assuming that Airplane, Car, and Boat are concrete, we apply Rule 1a to part a and we end up with the diagram in part b, where we have added the associations, the multiplicities, and the XOR constraint. If we apply Rule 1b to part a, we end up with the diagram in part c, where all the attributes of the superclasses have been copied down into Flying Car and Amphibious Car. In this latter case, you might have to deal with the effects of inheritance conflicts.

Implementing Problem-Domain Objects in a Traditional Language From a practical perspective, we are much better off implementing an object-oriented design in an object-oriented programming language, such as C++, Java, Objective-C, or Visual Basic.net. Practically speaking, the gulf between an object-oriented design and a traditional programming language is simply too great for mere mortals to be able to cross. The best advice that we can give about implementing an object-oriented design in a traditional programming language is to run away as fast and as far as possible from the project. However, if we are brave (foolish?) enough to attempt this, we must realize that in addition to factoring out inheritance from the design, we have to factor out all uses of polymorphism, dynamic binding, encapsulation, and information hiding. This is quite a bit of additional work to be accomplished. The way we factor these object-oriented features out of the detailed design of the system tends to be language dependent. This is beyond the scope of this text.

CONSTRAINTS AND CONTRACTS

Contracts were introduced in Chapter 5 in association with collaborations. A contract formalizes the interactions between the client and server objects, where a *client* (*consumer*) object is an instance of a class that sends a message to a *server* (*supplier*) object that executes one of its methods in response to the request. Contracts are modeled on the legal notion of a contract, where both parties, client and server objects, have obligations and rights. Practically speaking, a contract is a set of constraints and guarantees. If the constraints are met, then the server object guarantees certain behavior.[31] Constraints can be written in a natural language (e.g., English), a semiformal language (e.g., *Structured English*[32]), or a formal language (e.g., UML's Object Constraint Language). Given the need for precise, unambiguous specification of constraints, we recommend using UML's Object Constraint Language.

The *Object Constraint Language (OCL)*[33] is a complete language designed to specify constraints. In this section, we provide a short overview of some of the more useful constructs contained in the language (see Figure 8-18). Essentially, all OCL expressions are simply a declarative statement that evaluates to either being true or false. If the expression evaluates to true, then the constraint has been satisfied. For example, if a customer had to have a less than a one hundred dollar balance owed to be allowed to place another credit order, the OCL expression would be:

balance owed <= 100.00

OCL also has the ability to traverse relationships between objects, e.g., if the amount on a purchase order is required to be the sum of the values of the individual purchase order lines, this can be modeled as:

amount = OrderLine.sum(getPrice())

OCL also provides the ability to model more-complex constraints with a set of logical operators: and, or, xor, and not. For example, if customers were to be given a discount only if they were a senior citizen or a "prime" customer, OCL could be used to model the constraint as:

age > 65 or customerType = "prime"

[31] The idea of using contracts in design evolved from the "Design by Contract" technique developed by Bertrand Meyer. See Bertrand Meyer, *Object-Oriented Software Construction* (Englewood Cliffs, NJ: Prentice Hall, 1988).

[32] We describe Structured English with Method Specification later in this chapter.

[33] For a complete description of the object constraint language, see Jos Warmer and Anneke Kleppe, *The Object Constraint Language: Precise Modeling with UML* (Reading, MA: Addison-Wesley, 1999).

Operator Type	Operator	Example
Comparison	=	a = 5
	<	a < 100
	<=	a <= 100
	>	a > 100
	>=	a > = 100
	<>	a <> 100
Logical	and	a and b
	or	a or b
	xor	a xor b
	not	not a
Math	+	a + b
	−	a − b
	*	a * b
	/	a / b
String	concat	a = b.concat(c)
Relationship Traversal	.	relationshipAttributeName.b
	::	superclassName::propertyName
Collection	size	a.size
	count(object)	a.count(b)
	includes(object)	a.includes(b)
	isEmpty	a.isEmpty
	sum()	a.sum(b,c,d)
	select(expression)	a.select(b > d)

FIGURE 8-18
Sample OCL Constructs

OCL provides many other constructs that can be used to build unique constraints. These include math-oriented operators, string operators, and relationship traversal operators. For example, if the printed name on a customer order should be the concatenation of the customer's first name and last name, then OCL could represent this constraint as:

printedName = firstName.concat(lastName)

We already have seen an example of the '.' operator being used to traverse a relationship from Order to OrderLine above. The '::' operator allows the modeling of traversing inheritance relationships.

OCL also provides a set of operations that are used to support constraints over a collection of objects. For example, we demonstrated the use of the sum() operator above where we wanted to guarantee that the amount was equal to the summation of all of the prices of the items in the collection. The size operation returns the number of items in the collection. The count operation returns the number of occurrences in the collection of the specific object passed as its argument. The includes operation tests whether the object passed to it is already included in the collection. The isEmpty operation determines whether the collection is empty or not. The select operation provides support to model the identification of a subset of the collection based on the expression that is passed as its argument. Obviously, OCL provides a rich set of operators and operations in which to model constraints.

Types of Constraints

Three different types of constraints are typically captured in object-oriented design: preconditions, postconditions, and invariants.

Contracts are used primarily to establish the preconditions and postconditions for a method to be able to execute properly. A *precondition* is a constraint that must be met for a method to execute. For example, the parameters passed to a method must be valid for the method to execute. Otherwise, an exception should be raised. A *postcondition* is a constraint that must be met after the method executes, or the effect of the method execution must be undone. For example, the method cannot make any of the attributes of the object take on an invalid value. In this case, an exception should be raised, and the effect of the method's execution should be undone.

Whereas preconditions and postconditions model the constraints on an individual method, *invariants* model constraints that must always be true for all instances of a class. Examples of invariants include domains or types of attributes, multiplicity of attributes, and the valid values of attributes. This includes the attributes that model association and aggregation relationships. For example, if an association relationship is required, an invariant should be created that will enforce it to have a valid value for the instance to exist. Invariants are normally attached to the class. We can attach invariants to the CRC cards or class diagram by adding a set of assertions to them.

In Figure 8-19, the back of the CRC card constrains the attributes of an Order to specific types. For example, Order Number must be an unsigned long, and Customer must be an instance of the Customer class. Furthermore, additional invariants were added to four of the attributes. For example, Cust ID must not only be an unsigned long, but it also must have one and only one value [i.e., a multiplicity of (1..1)], and it must have the same value as the result of the GetCustID() message sent to the instance of Customer stored in the Customer attribute. Also shown is the constraint for an instance to exist, an instance of the Customer class, an instance of the State class, and at least one instance of the Product class must be associated with the Order object (see the Relationships section of the CRC card where the multiplicities are 1..1, 1..1, and 1..*, respectively). Figure 8-20 portrays the same set of invariants on a class diagram. However, if all invariants are placed on a class diagram, the diagram becomes very difficult to understand. Consequently, we recommend either extending the CRC card to document the invariants instead of attaching them all to the class diagram or creating a separate text document that contains them (see Figure 8-21).

Elements of a Contract

Contracts document the message passing that takes place between objects. Technically speaking, a contract should be created for each message sent and received by each object, one for each interaction. However, there would be quite a bit of duplication if this were done. In practice, a contract is created for each method that can receive messages from other objects (i.e., one for each visible method).

A contract should contain the information necessary for a programmer to understand what a method is to do (i.e., they are declarative in nature). This information includes the method name, class name, ID number, client objects, associated use cases, description, arguments received, type of data returned, and the pre- and postconditions.[34] Contracts do not

[34] Currently, there is no standard format for a contract. The contract in Figure 8-22 is based on material contained in Ian Graham, *Migrating to Object Technology* (Reading, MA: Addison-Wesley, 1995); Craig Larman, *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design* (Englewood Cliffs, NJ: Prentice Hall, 1998); Meyer, *Object-Oriented Software Construction;* R. Wirfs-Brock, B. Wilkerson, and L. Wiener, *Designing Object-Oriented Software* (Englewood Cliffs, NJ: Prentice Hall, 1990).

Front:

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

Responsibilities	**Collaborators**
Calculate subtotal	
Calculate tax	
Calculate shipping	
Calculate total	

(a)

Back:

Attributes:

Order Number	(1..1)	(unsigned long)	
Date	(1..1)	(Date)	
Sub Total	(0..1)	(double)	{Sub Total = ProductOrder. sum(GetExtension())}
Tax	(0..1)	(double)	(Tax = State.GetTaxRate() * Sub Total)
Shipping	(0..1)	(double)	
Total	(0..1)	(double)	
Customer	(1..1)	(Customer)	
Cust ID	(1..1)	(unsigned long)	{Cust ID = Customer. GetCustID()}
State	(1..1)	(State)	
StateName	(1..1)	(String)	{State Name = State. GetState()}

Relationships:

Generalization (a-kind-of):

Aggregation (has-parts):

Other Associations: Customer {1..1} State {1..1}Product {1..*}

(b)

FIGURE 8-19
Invariants on a CRC Card

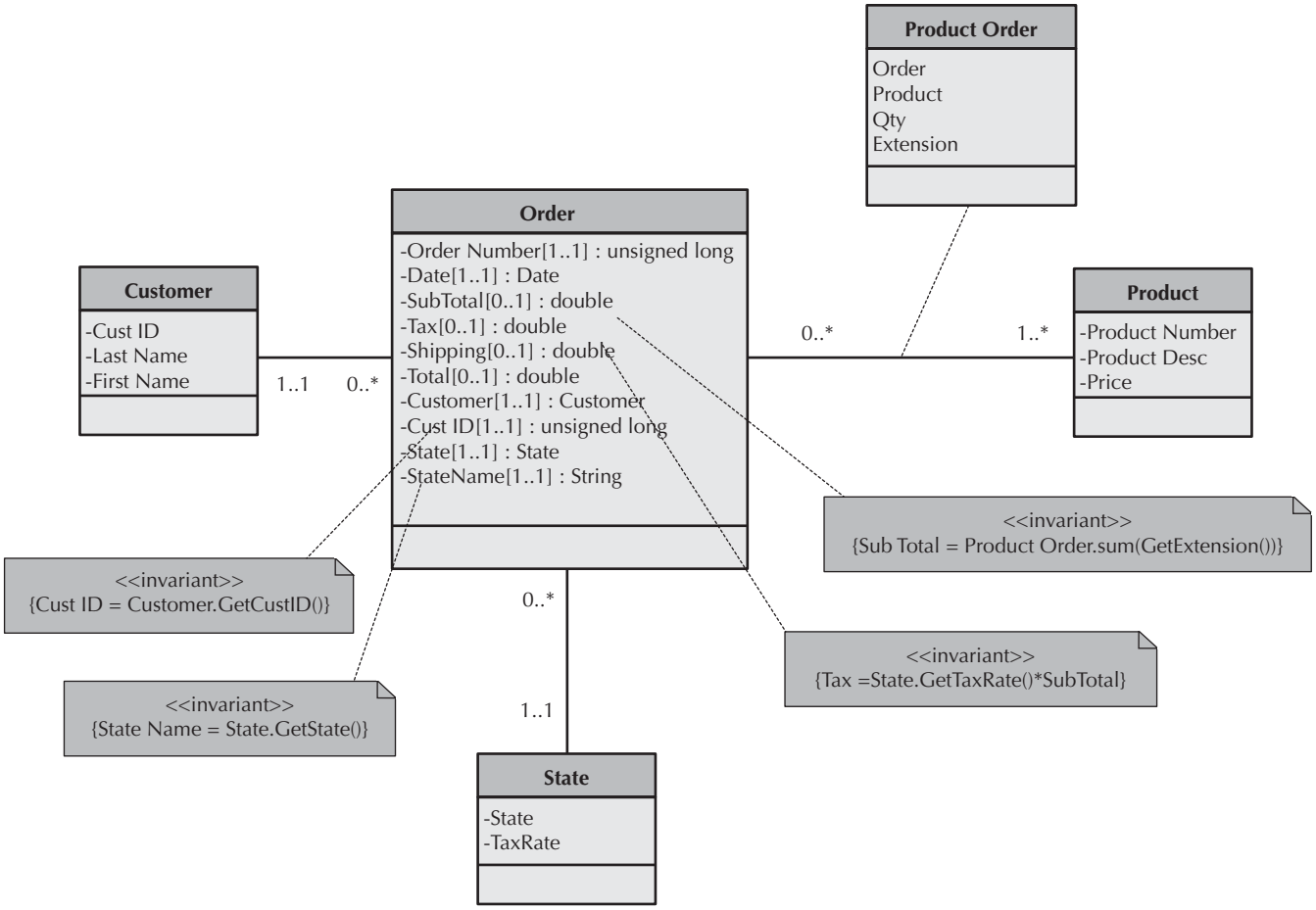


FIGURE 8-20 Invariants on a Class Diagram

have a detailed algorithmic description of how the method is to work. Detailed algorithmic descriptions typically are documented in a method specification (as described later in this chapter). In other words, a contract is composed of the information required for the developer of a client object to know what messages can be sent to the server objects and what the client can expect in return. Figure 8-22 shows a sample format for a contract.

Because each contract is associated with a specific method and a specific class, the contract must document them. The ID number of the contract is used to provide a unique identifier for every contract. The Clients (Consumers) element of a contract is a list of classes and

Order class invariants:

Cust ID = Customer.GetCustID()
State Name = Sate.GetState()
Sub Total = ProductOrder.sum(GetExtension())
Tax = State.GetTaxRate() * Sub Total1

FIGURE 8-21
Invariants in a Text File

Method Name:	Class Name:	ID:
Clients (Consumers):		
Associated Use Cases:		
Description of Responsibilities:		
Arguments Received:		
Type of Value Returned:		
Pre-Conditions:		
Post-Conditions:		

FIGURE 8-22
Sample Contract Form

methods that send a message to this specific method. This list is determined by reviewing the sequence diagrams associated with the server class. The Associated Use Cases element is a list of use cases in which this method is used to realize the implementation of the use case. The use cases listed here can be found by reviewing the server class's CRC card and the associated sequence diagrams.

The Description of Responsibilities provides an informal description of what the method is to perform, not how it is to do it. The arguments received are the data types of the parameters passed to the method, and the value returned is the data type of the value that the method returns to its clients. Together with the method name, they form the signature of the method.

The precondition and postcondition elements are where the pre- and postconditions for the method are recorded. Recall that pre- and postconditions can be written in a natural language, a semiformal language, or a formal language. As with invariants, we recommend that you use UML's Object Constraint Language.[35]

Example In this example, we return to the order example shown in Figures 8-15, 8-19, 8-20, and 8-21. In this case, we limit the discussion to the design of the addOrder method for the Customer class. The first decision we must make is how to specify the design of the relationship from Customer to Order. By reviewing Figures 8-15, 8-19, and 8-20, we see that the relationship has a multiplicity of 0..* which means that an instance of customer may exist without having any orders or an instance of customer could have many orders. As shown

[35] See Warmer and Kleppe, *The Object Constraint Language: Precise Modeling with UML*.

in Figure 8-15c, the relationship has been converted to an attribute that can contain many instances of the Order class.

However, an important question that would not typically come up during analysis is whether the order objects should be kept in sorted order or not. Another question that is necessary to have answered for design purposes is how many orders could be expected by a customer. The answers to these two questions will determine how we should organize the orders from the customer object's perspective. If the number of orders is going to be relatively small and the orders don't have to be kept in sorted order, then using a built-in programming language construct such as a vector is sufficient. However, if the number of orders is going to be large or the orders must be kept in sorted order, then some form of a sorted data structure, such as a linked list, is necessary. For example purposes, we assume that a customer's orders will need to be kept in sorted order and that there will be a large number of them. Therefore, instead of using a vector to contain the orders, we use a sorted singly linked list.

To keep the design of the Customer class as close to the problem domain representation as possible, the design of the Customer class is based on the Iterator pattern in Figure 8-13. For simplicity purposes, we assume that an order is created before it is associated with the specific customer. Otherwise, given the additional constraints of the instance of State class and the instance of the Product Order class existing before an instance of Order can be created would also have to be taken into consideration. This assumption allows us to ignore the fact that an instance of State can have many orders, an instance of Order can have many instances of Product Order associated with it, and an instance of Product can have many instances of Product Order associated with it, which would require us to design many additional containers (vectors or other data structures).

Based on all of the above, a new class diagram fragment was created that represents a linked list-based relationship between instances of the Customer class and instances of the Order class (see Figure 8-23). By carefully comparing Figures 8-15 and 8-23, we see that the Iterator pattern idea has been included between the Customer and Order classes. The domain of the Orders relationship-based attribute of the Customer class has been replaced with OrderList to show that the list of orders will be contained in a list data structure. Figure 8-24 portrays an object diagram-based representation of how the relationship between a customer instance and a set of order instances is stored in a sorted singly linked list data structure. In this case, we see that a Customer object has an OrderList object associated with it, each OrderList object could have N OrderNode objects, and each OrderNode object will have an Order object. We see that each Order object is associated with a single Customer object. By comparing Figures 8-15 and 8-24, we see that the intention of the multiplicity constraints of the Orders attribute of Customer, where a customer can have many orders, and the multiplicity constraints of the Customer attribute of Orders is being modeled correctly. Finally, notice that one of the operations contained in the OrderList class is a private method. We will return to this specific point in the next section that addresses method specification.

Using Figures 8-22, 8-23, and 8-24, contracts for the addOrder method of the Customer class and the insertOrder method for the OrderList class can be specified (see Figure 8-25). In the case of the addOrder method of the Customer class, we see that only instances of the Order class use the method (see Clients section), that the method only implements part of the logic that supports the addCustomerOrder use case (see Associated Use Cases section), and that the contract includes a short description of the methods responsibilities. We also see that the method receives a single argument of type Order and that it does not return anything (void). Finally, we see that both a precondition and a postcondition were specified. The precondition simply states that the new Order object cannot be included in the current list

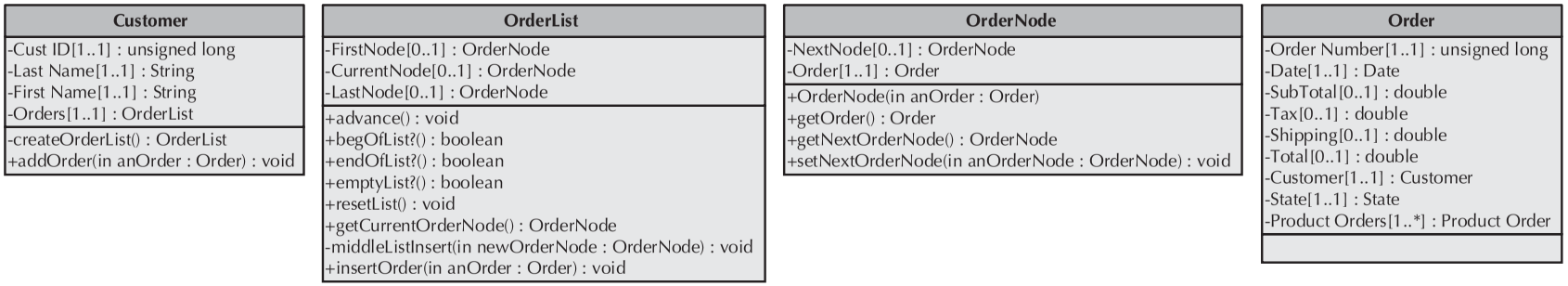


FIGURE 8-23 Class Diagram Fragment of the Customer to Order Relationship Modeled as a Sorted Singly Linked List

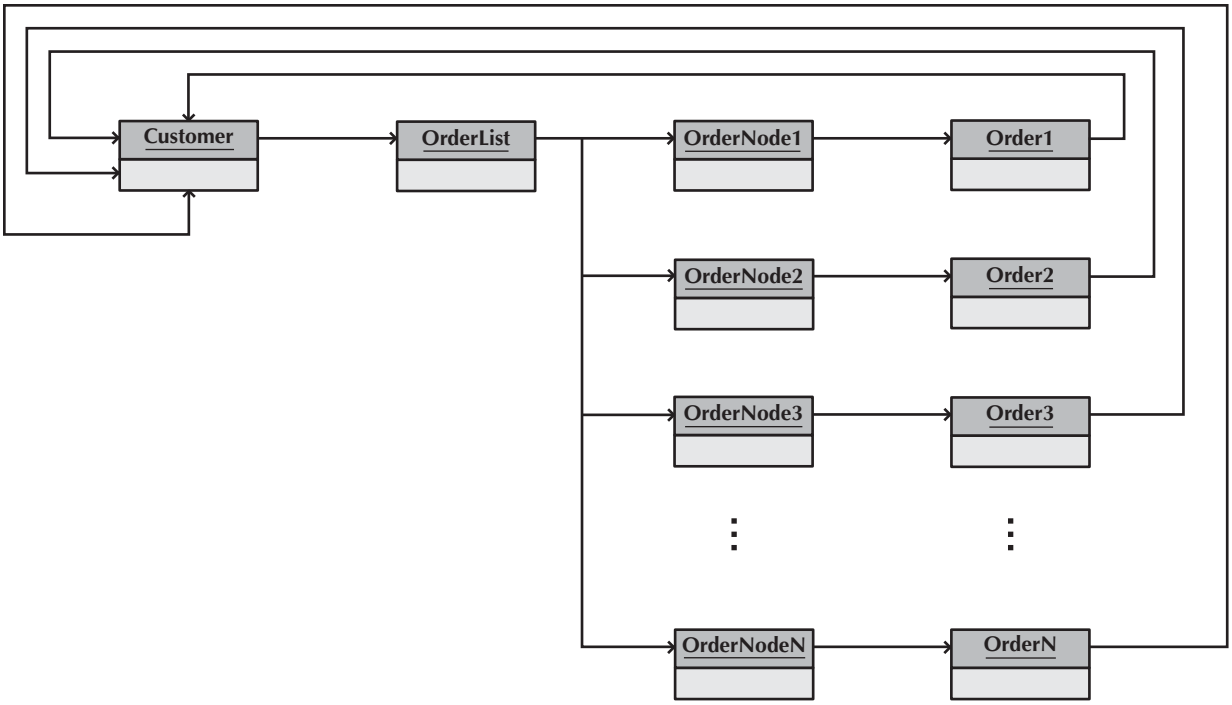


FIGURE 8-24 Object Diagram of the Customer to Order Relationship Modeled as a Sorted Singly Linked List

of Orders; that is, the order cannot have previously been associated with this customer. The postcondition, on the other hand, specifies that the new list of orders must be equal to the old list of orders (@pre) plus the new order object (including).

The contract for the insertOrder method for the OrderList class is somewhat simpler than the addOrder method's contract. From a practical perspective, the insertOrder method implements part of the addOrder method's logic. Specifically speaking, it implements that actual insertion of the new order object into the specific data structure chosen to manage the list of Order objects associated with the specific Customer object. Consequently, because we already have specified the precondition and postcondition for the addOrder method, we do not have to further specify the same constraints for the insertOrder method. However, this does implicitly increase the dependence of the Customer objects on the implementation chosen for the list of customer orders. This is a good example of moving from the problem domain to the solution domain. While we were focusing on the problem domain during analysis, the actual implementation of the list of orders was never considered. However, because we now are designing the implementation of the relationship between the Customer objects and the Order objects, we have had to move away from the language of the end user and toward the language of the programmer. During design, the focus moves toward optimizing the code to run faster on the computer and not worrying about the end user's ability to understand the inner workings of the system; from an end user's perspective, the system should become more of a black box with which they interact. As we move farther into the detailed design of the implementation of the problem domain classes, some solution domain classes, such as the approach to implement relationships, will creep into the specification of

Method Name: addOrder	**Class Name:** Customer	**ID:** 36
Clients (consumers): Order		
Associated Use Cases: addCustomerOrder		
Description of Responsibilities: Implement the necessary behavior to add a new order to an existing customer keeping the orders in sorted order by the order's order number.		
Arguments Received: anOrder:Order		
Type of Value Returned: void		
Pre-Conditions: not orders.includes(anOrder)		
Post-Conditions: Orders = Orders@pre.including(anOrder)		

Method Name: insertOrder	**Class Name:** OrderList	**ID:** 123
Clients (consumers): Customer		
Associated Use Cases: addCustomerOrder		
Description of Responsibilities: Implement inserting an Order object into an OrderNode object and manage the insertion of the OrderNode object into the current location in the sorted singly linked list of orders.		
Arguments Received: anOrder:Order		
Type of Value Returned: void		
Pre-Conditions: None.		
Post-Conditions: None.		

FIGURE 8-25
Sample Contract for the addOrder Method of the Customer Class and the insertOrder Method of the OrderList Class

the problem domain layer. In this particular example, the OrderList and OrderNode classes also could be used to implement the relationships from State objects to Order objects, from Order Objects to Product Order objects, and from Product objects to Product Order objects (see Figure 8-15). Given our simple example, one can clearly see that specifying the design of the problem domain layer could include many additional solution domain classes to be specified on the problem domain layer.

METHOD SPECIFICATION

Once the analyst has communicated the big picture of how the system needs to be put together, he or she needs to describe the individual classes and methods in enough detail so that programmers can take over and begin writing code. Methods on the CRC cards, class diagram, and contracts are described using *method specifications*. Method specifications are written documents that include explicit instructions on how to write the code to implement the method. Typically, project team members write a specification for each method and then pass them all along to programmers who write the code during implementation of the project. Specifications need to be very clear and easy to understand, or programmers will be slowed down trying to decipher vague or incomplete instructions.

There is no formal syntax for a method specification, so every organization uses its own format, often using a form like the one in Figure 8-26. Typical method specification forms contain four components that convey the information that programmers will need for writing the appropriate code: general information, events, message passing, and algorithm specification.

General Information

The top of the form in Figure 8-26 contains general information, such as the name of the method, name of the class in which this implementation of the method will reside, ID number, Contract ID (which identifies the contract associated with this method implementation), programmer assigned, the date due, and the target programming language. This information is used to help manage the programming effort.

Events

The second section of the form is used to list the events that trigger the method. An *event* is a thing that happens or takes place. Clicking the mouse generates a mouse event, pressing a key generates a keystroke event—in fact, almost everything the user does generates an event.

In the past, programmers used procedural programming languages that contained instructions that were implemented in a predefined order, as determined by the computer system, and users were not allowed to deviate from the order. Many programs today are *event driven* (e.g., programs written in languages such as Visual Basic, Objective C, C++, or Java), and event-driven programs include methods that are executed in response to an event initiated by the user, system, or another method. After initialization, the system waits for an event to occur. When it does, a method is fired that carries out the appropriate task, and then the system waits once again.

We have found that many programmers still use method specifications when programming in event-driven languages, and they include the event section on the form to capture when the method will be invoked. Other programmers have switched to other design tools that capture event-driven programming instructions, such as the behavioral state machine described in Chapter 6.

Method Name:	Class Name:	ID:
Contract ID:	Programmer:	Date Due:
Programming Language: ❑ Visual Basic ❑ Smalltalk ❑ C++ ❑ Java		
Triggers/Events:		

Arguments Received: Data Type:	Notes:

Messages Sent & Arguments Passed: ClassName.MethodName:	Data Type:	Notes:

Arguments Returned: Data Type:	Notes:

Algorithm Specification:

Misc. Notes:

FIGURE 8-26
Method Specification Form

Message Passing

The next sections of the method specification describe the message passing to and from the method, which are identified on the sequence and collaboration diagrams. Programmers need to understand what arguments are being passed into, passed from, and returned by the method because the arguments ultimately translate into attributes and data structures within the actual method.

Common Statements	Example
Action Statement	Profits = Revenues – Expenses Generate Inventory-Report
If Statement	IF Customer Not in the Customer Object Store THEN Add Customer record to Customer Object Store ELSE Add Current-Sale to Customer's Total-Sales Update Customer record in Customer Object Store
For Statement	FOR all Customers in Customer Object Store DO Generate a new line in the Customer-Report Add Customer's Total-Sales to Report-Total
Case Statement	CASE IF Income < 10,000: Marginal-tax-rate = 10 percent IF Income < 20,000: Marginal-tax-rate = 20 percent IF Income < 30,000: Marginal-tax-rate = 31 percent IF Income < 40,000: Marginal-tax-rate = 35 percent ELSE Marginal-Tax-Rate = 38 percent ENDCASE

FIGURE 8-27
Structured English

Algorithm Specifications

Algorithm specifications can be written in Structured English or some type of formal language.[36] Structured English is simply a formal way of writing instructions that describe the steps of a process. Because it is the first step toward the implementation of the method, it looks much like a simple programming language. Structured English uses short sentences that clearly describe exactly what work is performed on what data. There are many versions of Structured English because there are no formal standards; each organization has its own type of Structured English. Figure 8-27 shows some examples of commonly used Structured English statements.

Action statements are simple statements that perform some action. An If statement controls actions that are performed under different conditions, and a For statement (or a While statement) performs some actions until some condition is reached. A Case statement is an advanced form of an If statement that has several mutually exclusive branches.

If the algorithm of a method is complex, a tool that can be useful for algorithm specification is UML's *activity diagram* (see Figure 8-28 and Chapter 4). Recall that activity diagrams can be used to specify any type of process. Obviously, an algorithm specification represents a process. However, owing to the nature of object orientation, processes tend to be highly distributed over many little methods over many objects. Needing to use an activity diagram to specify the algorithm of a method can, in fact, hint at a problem in the design. For example, the method should be further decomposed or there could be missing classes.

The last section of the method specification provides space for other information that needs to be communicated to the programmer, such as calculations, special business rules, calls to subroutines or libraries, and other relevant issues. This also can point out

[36] For our purposes, Structured English will suffice. However, there has been some work with the Catalysis, Fusion, and Syntropy methodologies to include formal languages, such as VDM and Z, into specifying object-oriented systems.

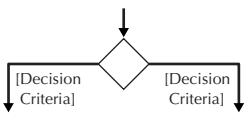
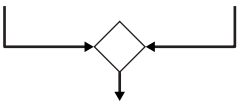
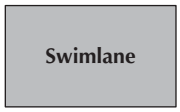
An action: ■ Is a simple, nondecomposable piece of behavior. ■ Is labeled by its name.	
An activity: ■ Is used to represent a set of actions. ■ Is labeled by its name.	
An object node: ■ Is used to represent an object that is connected to a set of object flows. ■ Is labeled by its class name.	
A control flow: ■ Shows the sequence of execution.	
An object flow: ■ Shows the flow of an object from one activity (or action) to another activity (or action).	
An initial node: ■ Portrays the beginning of a set of actions or activities.	
A final-activity node: ■ Is used to stop all control flows and object flows in an activity (or action).	
A final-flow node: ■ Is used to stop a specific control flow or object flow.	
A decision node: ■ Is used to represent a test condition to ensure that the control flow or object flow only goes down one path. ■ Is labeled with the decision criteria to continue down the specific path.	
A merge node: ■ Is used to bring back together different decision paths that were created using a decision node.	
A Fork node: Is used to split behavior into a set of parallel or concurrent flows of activities (or actions).	
A Join node: Is used to bring back together a set of parallel or concurrent flows of activities (or actions).	
A Swimlane: Is used to break up an activity diagram into rows and columns to assign the individual activities (or actions) to the individuals or objects that are responsible for executing the activity (or action). Is labeled with the name of the individual or object responsible.	

FIGURE 8-28 Syntax for an Activity Diagram (Figure 4-7)

changes or improvements that will be made to any of the other design documentation based on problems that the analyst detected during the specification process.[37]

Example

This example continues the addition of a new order for a customer described in the previous section (see Figure 8-29). Even though in most cases, because there are libraries

Method Name: insertOrder	**Class Name:** OrderList	**ID:** 100
Contract ID: 123	**Programmer:** J. Doe	**Date Due:** 1/1/12

Programming Language:

❑ **Visual Basic** ❑ **Smalltalk** ❑ **C++** ❑ **Java**

Triggers/Events:

Customer places an order

Arguments Received: **Data Type:**	**Notes:**
Order	The new customer's new order.

Messages Sent & Arguments Passed: **ClassName.MethodName:**	**Data Type:**	**Notes:**
OrderNode.new()	Order	
OrderNode.getOrder()		
Order.getOrderNumber()		
OrderNode.setNextNode()	OrderNode	
self.middleListInsert()	OrderNode	

Arguments Returned: **Data Type:**	**Notes:**
void	

Algorithm Specification:

See Figures 8-30 and 8-31.

Misc. Notes:

None.

FIGURE 8-29 Method Specification for the insertOrder Method

[37] Remember that the development process is very incremental and iterative. Therefore, changes could be cascaded back to any point in the development process (e.g., to use-case descriptions, use-case diagrams, CRC cards, class diagrams, object diagrams, sequence diagrams, communication diagrams, behavioral state machines, and package diagrams).

```
Create new OrderNode with the new Order
IF emptyList?()
   FirstNode = LastNode = CurrentNode = newOrderNode
ELSE IF newOrderNode.getOrder().getOrderNumber() < FirstNode.getOrder().getOrderNumber()
   newOrderNode.setNextNode(FirstNode)
   FirstNode = newOrderNode
ELSE IF newOrderNode.getOrder().getOrderNumber() > LastNode.getOrder().getOrderNumber()
   LastNode.setNextNode(newOrderNode)
   LastNode = newOrderNode
ELSE
   middleListInsert(newOrderNode)
```

FIGURE 8-30
Structured English-based Algorithm Specification for the insertOrder Method

of data structure classes available that you could simply reuse and therefore would not need to specify the algorithm to insert into a sorted singly linked list, we use it as an example of how method specification can be accomplished. The general information section of the specification documents the method's name, its class, its unique ID number, the ID number of its associated contract, the programmer assigned, the date that its implementation is due, and the programming language to be used. Second, the trigger/event that caused this method to be executed is identified. Third, the data type of the argument passed to this method is documented (Order). Fourth, owing to the overall complexity of inserting a new node into the list, we have factored out one specific aspect of the algorithm into a separate private method (middleListInsert()) and we have specified that this method will be sending messages to instances of the OrderNode class and the Order class. Fifth, we specify the type of return value that insertOrder will produce. In this case, the insertOrder method will not return anything (void). Finally, we specify the actual algorithm. In this example, for the sake of completeness, we provide both a Structured English–based (see Figure 8-30) and an activity diagram–based algorithm specification (see Figure 8-31). Previously, we stated that we had factored out the logic of inserting into the middle of the list into a separate private method: middleListInsert(). Figure 8-32 shows the logic of this method. Imagine collapsing this logic back into the logic of the insertOrder method, i.e., replace the middleListInsert(newOrderNode) activity in Figure 8-31 with the contents of Figure 8-32. Obviously, the insertOrder method would be more complex.

VERIFYING AND VALIDATING CLASS AND METHOD DESIGN

Like all of the previous problem domain models, the constraints, contracts, and method specifications need to be verified and validated. Given that we are primarily dealing with the problem domain in this chapter, the constraints and contracts were derived from the functional requirements and the problem domain representations. However, they are applicable to the other layers. In that case, they would be derived from the solution domain representations associated with the data management (Chapter 9), human–computer interaction (Chapter 10), and system architecture (Chapter 11) layers. Given all of the issues described earlier with the design criteria (coupling, cohesion, and connascence), additional specifications, reuse opportunities, design restructuring and optimization, and mapping to implementation languages, it is likely that many modifications have taken place to the analysis representations of the problem domain. Consequently, virtually everything must be re-verified and re-validated.

First, we recommend that a walkthrough of all of the evolved problem domain representations be performed. That is, all functional models (Chapter 4) must be consistent; all structural

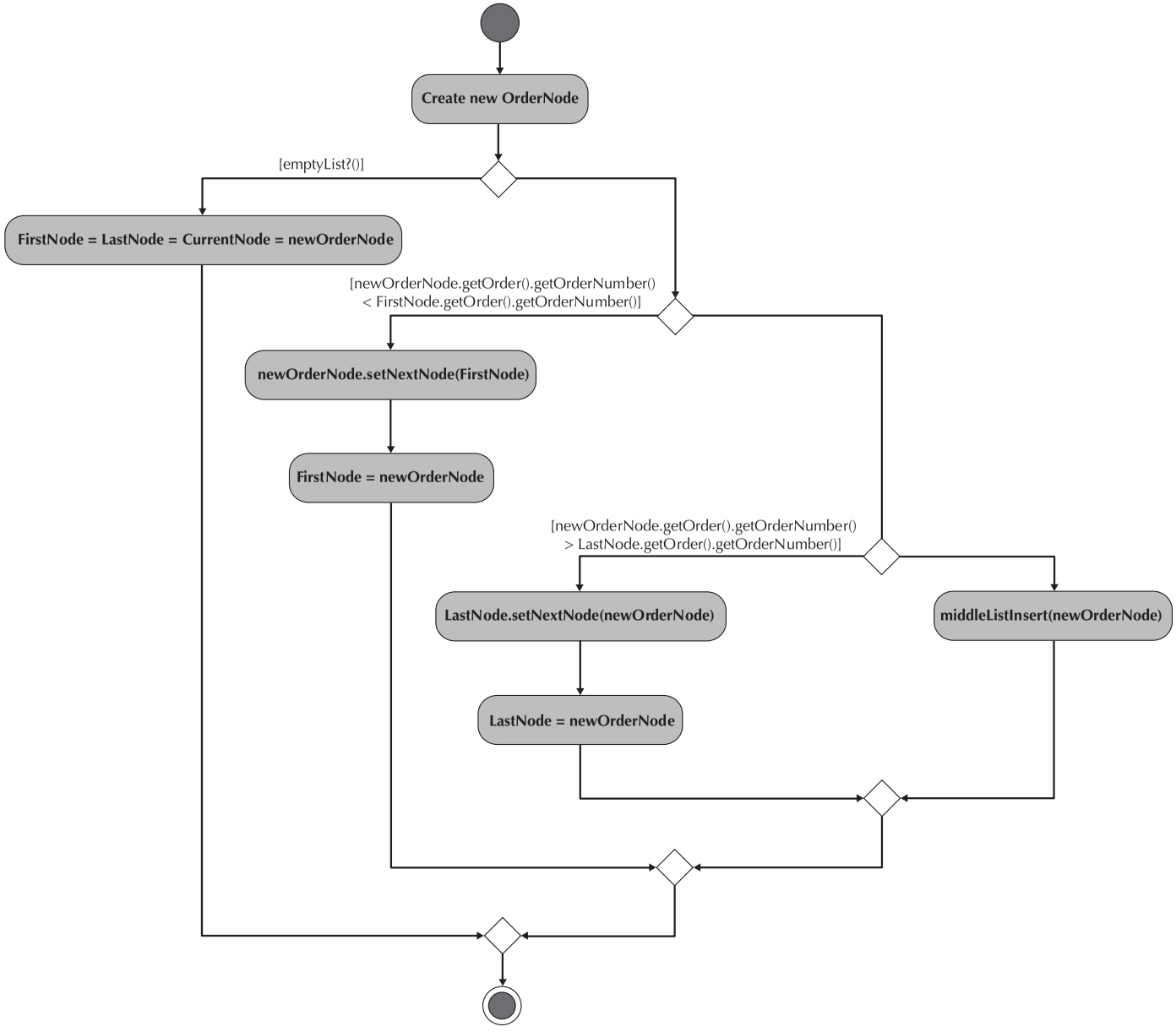


FIGURE 8-31 Activity Diagram-based Algorithm Specification for the insertOrder Method

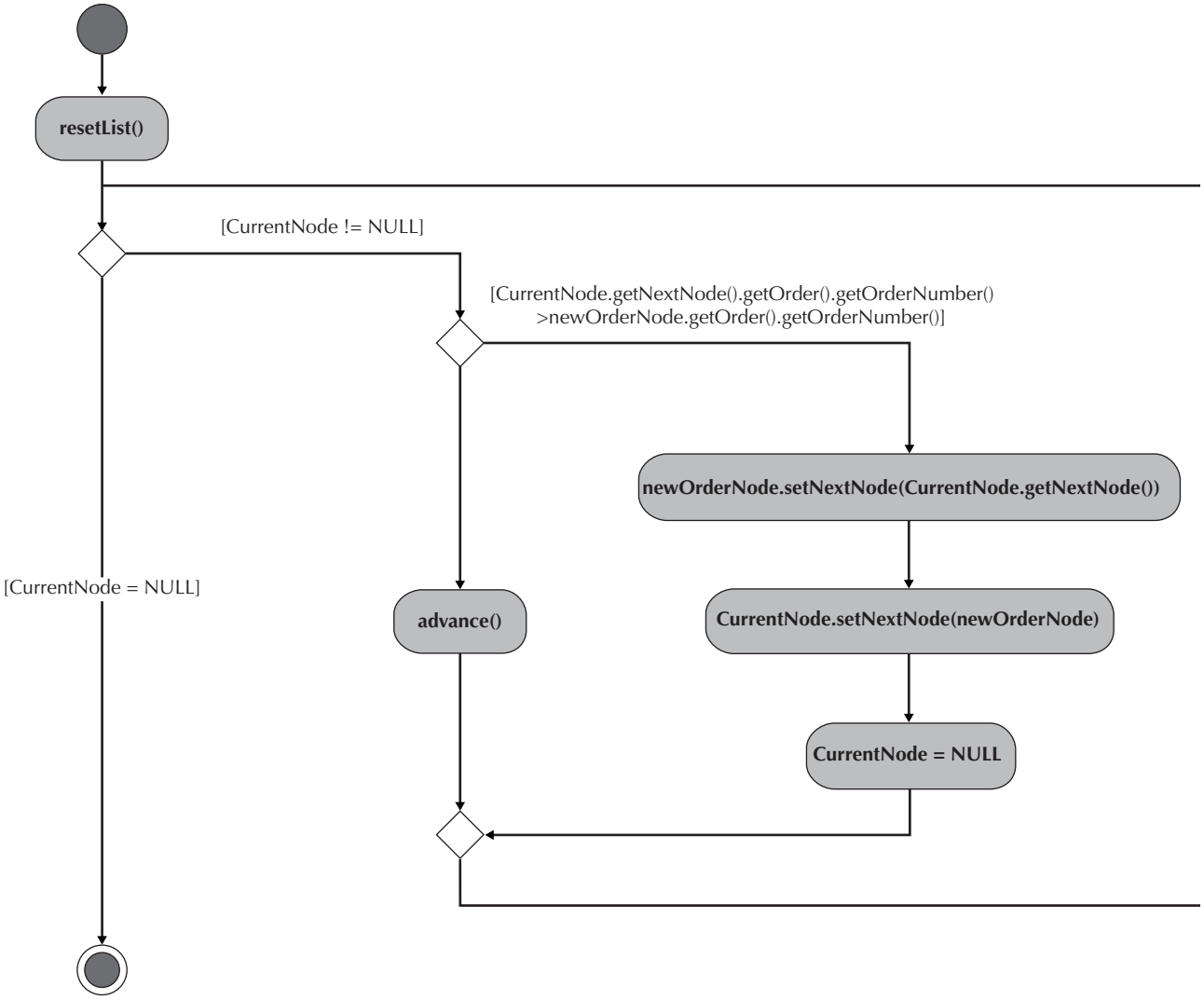


FIGURE 8-32 Activity Diagram–based Algorithm Specification for the middleListInsert Method

models (Chapter 5) must be consistent; all behavioral models (Chapter 6) must be consistent; and the functional, structural, and behavioral models must be balanced (Chapter 7).

Second, all constraints, contracts, and method specifications must be tested. The best way to do this is to role-play the system using the different scenarios of the use cases. In this case, we must enforce the invariants on the evolved CRC cards (see Figure 8-19), the pre- and post-conditions on the contract forms (see Figures 8-22 and 8-25), and the design of each method specified with the method specification forms (see Figures 8-26 and 8-29) and algorithm specifications (see Figures 8-30, 8-31, and 8-32).

Given the amount of verifying and validating the fidelity of all of the models that we have performed on the evolving system, it might seem like overkill to perform the above again. However, given the pure volume of changes that can take place during design, it is crucial to thoroughly test the models again before the system is implemented. In fact, testing is so important to the agile development approaches, testing forms the virtual backbone of those methodologies. Without thorough testing, there is no guarantee that the system being implemented will address the problem being solved. Once the system has been implemented, testing becomes even more important (see Chapter 12).

APPLYING THE CONCEPTS AT PATTERSON SUPERSTORE

In this installment of the Patterson Superstore case, Ruby and her team have shifted their focus from capturing the requirements, behavior, and structure of the evolving system to the design of the individual classes and method for the system. First, they had to return once more to the functional, structural, and behavior models to ensure that the classes defined in analysis (the problem domain layer) are both sufficient and necessary. In evaluating these models, they checked for coupling, cohesion, and connascence. They then moved to designing the contracts and method specifications.

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 basic characteristics of object orientation.
- ☐ Describe the problems that can arise when using polymorphism and inheritance.
- ☐ Describe the different types of inheritance conflicts.
- ☐ Describe the different types of coupling and why coupling should be minimized.
- ☐ Describe the law of Demeter.
- ☐ Describe the different types of cohesion and why cohesion should be maximized.
- ☐ Describe connascence.
- ☐ Identify opportunities for reuse through the use of patterns, frameworks, class libraries, and components.
- ☐ Optimize a design.
- ☐ Map the problem domain classes to a single-inheritance language.
- ☐ Map the problem domain classes to an object-based language.
- ☐ Understand the difficulties in implementing an object-oriented design in a traditional programming language.
- ☐ Use the OCL to define precondition, postcondition, and invariant constraints.
- ☐ Create contracts to specify the interaction between client and server objects.
- ☐ Specify methods using the method specification form.
- ☐ Specify the logic of a method using Structured English and activity diagrams.
- ☐ Understand how to verify and validate both the design of the classes and the design of their methods.

KEY TERMS

Active value
Activity diagram
API (application program interface)
Attribute
Behavior
Class
Class cohesion
Class library
Client
Cohesion
Component
Connascence
Constraint
Consumer
Contract
Coupling
Derived attribute
Design pattern
Dynamic binding
Encapsulation
Event
Event driven
Exceptions
Factoring
Fan-out
Framework
Generalization/specialization cohesion
Heteronyms
Homographs
Homonyms
Ideal class cohesion
Information hiding
Inheritance
Inheritance conflict
Inheritance coupling
Instance
Interaction coupling
Invariant
Law of Demeter
Message
Method
Method cohesion
Method specification
Multiple inheritance
Normalization
Object
Object-based language
Object constraint language (OCL)
Operations

Patterns	Redefinition	State	Trigger
Polymorphism	Server	Structured English	Visibility
Postcondition	Signature	Supplier	
Precondition	Single inheritance	Synonyms	

QUESTIONS

1. What are the basic characteristics of object-oriented systems?
2. What is dynamic binding?
3. Define polymorphism. Give one example of a good use of polymorphism and one example of a bad use of polymorphism.
4. What is an inheritance conflict? How does an inheritance conflict affect the design?
5. Why is cancellation of methods a bad thing?
6. Give the guidelines to avoid problems with inheritance conflicts.
7. Why is it important to know which object-oriented programming language is going to be used to implement the system?
8. What additional types of inheritance conflicts are there when using multiple inheritance?
9. What is the law of Demeter?
10. What are the six types of interaction coupling? Give one example of good interaction coupling and one example of bad interaction coupling.
11. What are the seven types of method cohesion? Give one example of good method cohesion and one example of bad method cohesion.
12. What are the four types of class cohesion? Give one example of each type.
13. What are the five types of connascence described in your text? Give one example of each type.
14. When designing a specific class, what types of additional specification for a class could be necessary?
15. What are exceptions?
16. What are constraints? What are the three different types of constraints?
17. What are patterns, frameworks, class libraries, and components? How are they used to enhance the evolving design of the system?
18. How are factoring and normalization used in designing an object system?
19. What are the different ways to optimize an object system?
20. What is the typical downside of system optimization?
21. What is the purpose of a contract? How are contracts used?
22. What is the Object Constraint Language? What is its purpose?
23. What is the Structured English? What is its purpose?
24. What is an invariant? How are invariants modeled in a design of a class? Give an example of an invariant for an hourly employee class using the Object Constraint Language.
25. Create a contract for a compute pay method associated with an hourly employee class. Specify the preconditions and postconditions using the Object Constraint Language.
26. How do you specify a method's algorithm? Give an example of an algorithm specification for a compute pay method associated with an hourly employee class using Structured English.
27. How do you specify a method's algorithm? Give an example of an algorithm specification for a compute pay method associated with an hourly employee class using an activity diagram.
28. How are methods specified? Give an example of a method specification for a compute pay method associated with an hourly employee class.

EXERCISES

A. For the A Real Estate Inc. problem in Chapters 4 (exercises I, J, and K), 5 (exercises P and Q), 6 (exercise D), and 7 (exercise A):

1. Choose one of the classes and create a set of invariants for attributes and relationships and add them to the CRC card for the class.
2. Choose one of the methods in the class that you chose and create a contract and a method specification for it. Use OCL to specify any pre- or postcondition and use both Structured English and an activity diagram to specify the algorithm.

B. For the A Video Store problem in Chapters 4 (exercises L, M, N K), 5 (exercises R and S), 6 (exercise E), and 7 (exercise B):

1. Choose one of the classes and create a set of invariants for attributes and relationships and add them to the CRC card for the class.
2. Choose one of the methods in the class that you chose and create a contract and a method specification for it. Use OCL to specify any pre- or postcondition and use both Structured English and an activity diagram to specify the algorithm.

C. For the gym membership problem in Chapters 4 (exercises O, P, and Q), 5 (exercises T and U), 6 (exercise F), and 7 (exercise C):

1. Choose one of the classes and create a set of invariants for attributes and relationships and add them to the CRC card for the class.
2. Choose one of the methods in the class that you chose and create a contract and a method specification for it. Use OCL to specify any pre- or postcondition and use both Structured English and an activity diagram to specify the algorithm.

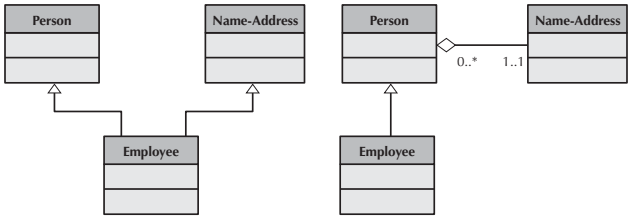
D. For the Picnics R Us problem in Chapters 4 (exercises R, S, and T), 5 (exercises V and W), 6 (exercise G), and 7 (exercise D):

1. Choose one of the classes and create a set of invariants for attributes and relationships and add them to the CRC card for the class.
2. Choose one of the methods in the class that you chose and create a contract and a method specification for it. Use OCL to specify any pre- or postcondition and use both Structured English and an activity diagram to specify the algorithm.

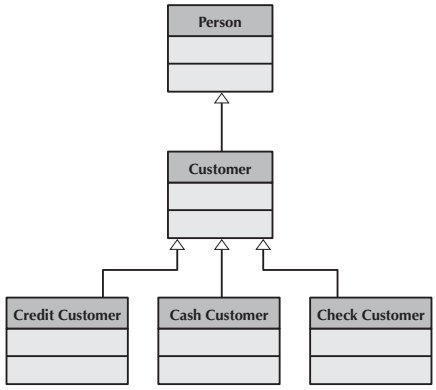
E. For the Of-the-Month-Club problem in Chapters 4 (exercises U, V, and W), 5 (exercises X and Y), 6 (exercise H), and 7 (exercise E):

1. Choose one of the classes and create a set of invariants for attributes and relationships and add them to the CRC card for the class.
2. Choose one of the methods in the class that you chose and create a contract and a method specification for it. Use OCL to specify any pre- or postcondition and use both Structured English and an activity diagram to specify the algorithm.

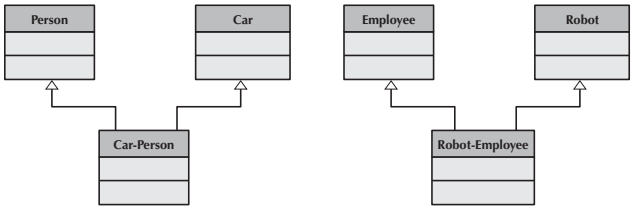
F. Describe the difference in meaning between the following two class diagrams. Which is a better model? Why?



G. From a cohesion, coupling, and connascence perspective, is the following class diagram a good model? Why or why not?



H. From a cohesion, coupling, and connascence perspective, are the following class diagrams good models? Why or why not?



I. Create a set of inheritance conflicts for the two inheritance structures in the class diagrams of exercise H.

MINICASES

1. Your boss has been in the software development field for thirty years. He has always prided himself on his ability to adapt his skills from one approach to developing software to the next approach. For example, he had no problem learning structured analysis and design in the early 1980s and information

engineering in the early 1990s. He even understands the advantage of rapid application development. But the other day, when you and he were talking about the advantages of object-oriented approaches, he became totally confused. He thought that characteristics such as polymorphism and inheritance were an advantage for object-oriented systems. However, when you explained the problems with inheritance conflicts, redefinition capabilities, and the need for semantic consistency across different implementations of methods, he was ready to simply give up. To make matters worse, you then went on to explain the importance of contracts in controlling the development of the system. At this point in the conservation, he basically threw in the towel. As he walked off, you heard him say something like "I guess it's true, it's too hard to teach an old dog new tricks."

Being a loyal employee and friend, you decided to write a short tutorial to give your boss on object-oriented systems development. As a first step, create a detailed outline for the tutorial. As a subtle example, use good design criteria, such as coupling and cohesion, in the design of your tutorial outline.

2. You have been working with the professional and scientific management (PSSM) problem for quite a while. You should go back and refresh your memory about the problem before attempting to solve this situation. Refer back to your solutions to Minicase 3 in Chapter 7.

a. For each class in the structural model, using OCL, create a set of invariants for attributes and relationships and add them to the CRC cards for the classes.

b. Choose one of the classes in the structural model. Create a contract for each method in that class. Be sure to use OCL to specify the preconditions and the postconditions. Be as complete as possible.

c. Create a method specification for each method in the class you chose for question b. Use both Structured English and activity diagrams for the algorithm specification.

3. You have been working with the Holiday Travel Vehicle problem for quite a while. You should go back and refresh your memory about the problem before attempting to solve this situation. Refer back to your solutions Minicase 4 in Chapter 7.

In the new system for Holiday Travel Vehicles, the system users follow a two-stage process to record complete information on all of the vehicles sold. When an RV or trailer first arrives at the company from the manufacturer, a clerk from the inventory department creates a new vehicle record for it in the computer system. The data entered at this time include basic descriptive information on the vehicle such as manufacturer, name, model, year, base cost, and freight charges. When the vehicle is sold, the new vehicle record is updated to reflect the final sales terms and the dealer-installed options added to the vehicle. This information is entered into the system at the time of sale when the salesperson completes the sales invoice.

When it is time for the clerk to finalize the new vehicle record, the clerk selects a menu option from the system, which is called Finalize New Vehicle Record. The tasks involved in this process are described below.

When the user selects Finalize New Vehicle Record from the system menu, the user is immediately prompted for the serial number of the new vehicle. This serial number is used to retrieve the new vehicle record for the vehicle from system storage. If a record cannot be found, the serial number is probably invalid. The vehicle serial number is then used to retrieve the option records that describe the dealer-installed options that were added to the vehicle at the customer's request. There may be zero or more options. The cost of the option specified on the option record(s) is totaled. Then, the dealer cost is calculated using the vehicle's base cost, freight charge, and total option cost. The completed new vehicle record is passed back to the calling module.

a. Update the structural model (CRC cards and class diagram) with this additional information.

b. For each class in the structural model, using OCL, create a set of invariants for attributes and relationships and add them to the CRC cards for the classes.

c. Choose one of the classes in the structural model. Create a contract for each method in that class. Be sure to use OCL to specify the preconditions and the postconditions. Be as complete as possible.

d. Create a method specification for each method in the class you chose for question b. Use both Structured English and activity diagrams for the algorithm specification.