

2024

Tarlier

Chinh Thúc

PART TWO

DESIGN MODELING

Whereas analysis modeling concentrated on the functional requirements of the evolving system, design modeling incorporates the nonfunctional requirements. That is, design modeling focuses on *how* the system will operate. First, the project team verifies and validates the analysis models (functional, structural, and behavioral). Next, a set of factored and partitioned analysis models are created. The class and method designs are illustrated using the class specifications (using CRC cards and class diagrams), contracts, and method specifications. Next, the data management layer is addressed by designing the actual database or file structure to be used for object persistence, and a set of classes that will map the class specifications into the object persistence format chosen. Concurrently, the team produces the user interface layer design using use scenarios, windows navigation diagrams, real use cases, interface templates, storyboards, windows layout diagrams, and user interface prototypes. The physical architecture layer design is created using deployment diagrams and hardware software specifications. This collection of deliverables represents the system specification that is handed to the programming team for implementation.

CHAPTER 7

Moving on to Design

CHAPTER 8

Class and Method Design

CHAPTER 9

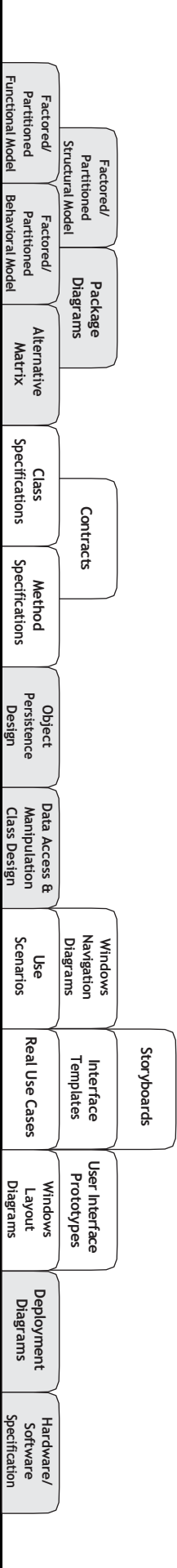
Data Management Layer Design

CHAPTER 10

Human Computer Interaction Layer Design

CHAPTER 11

Physical Architecture Layer Design



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

Avoiding Classic Design

TIP

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, 1996).

VERIFYING AND VALIDATING THE ANALYSIS MODELS¹

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.

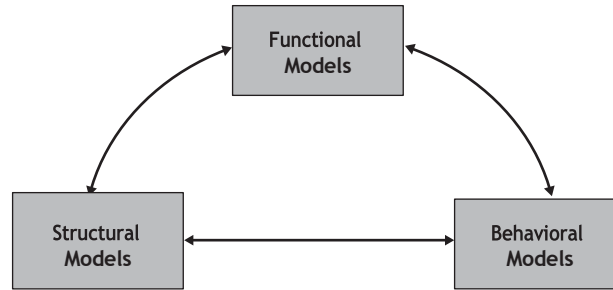
Second, every activity or action contained in an activity diagram (see Figure 7-6) and every event contained in a use-case description (see Figure 7-5) should be related to one or more responsibilities on a CRC card and one or more operations in a class on a class diagram and vice versa. For example, 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.

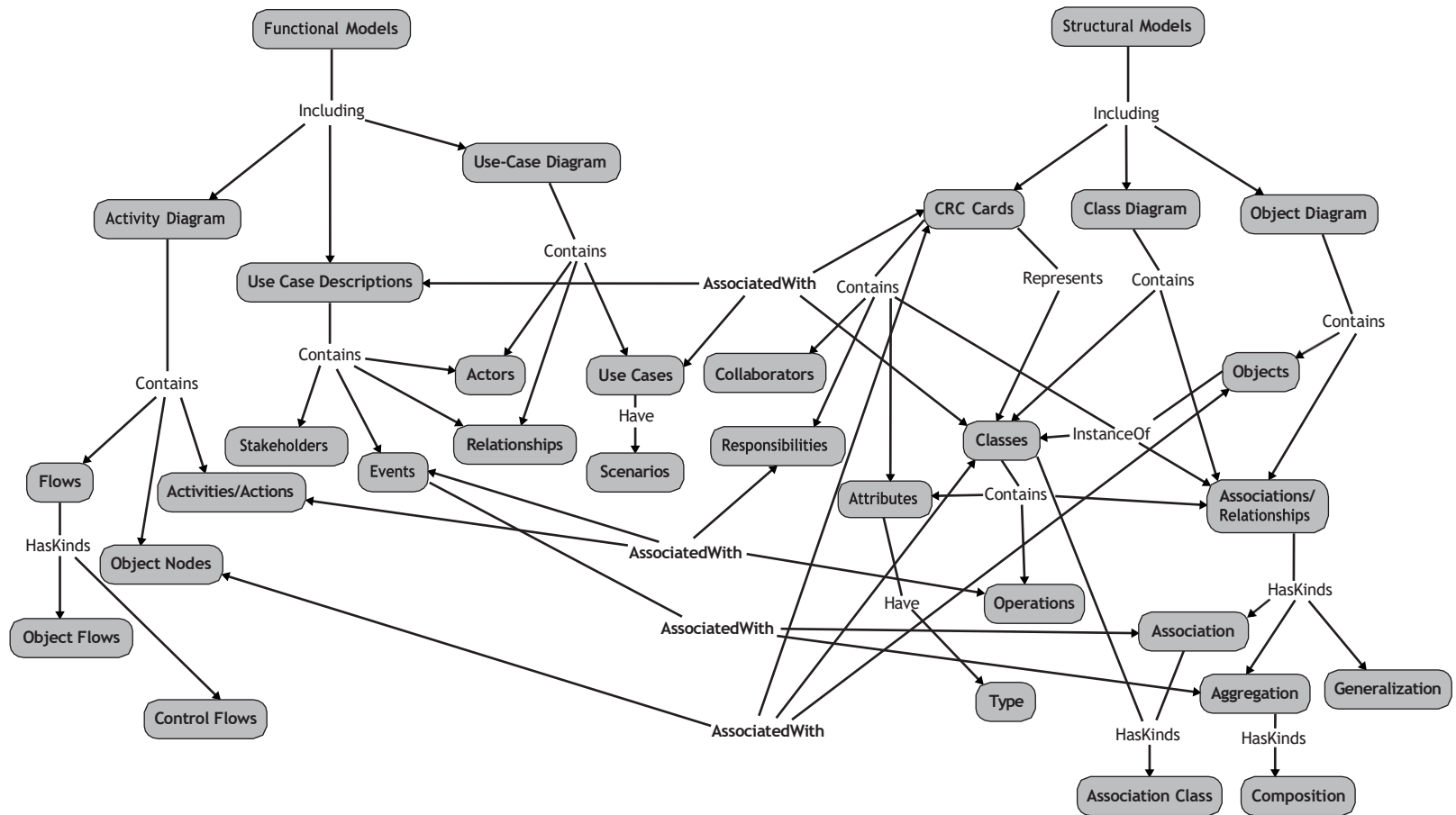


FIGURE 7-2 Relationships among Functional and Structural Models

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
<div style="text-align: center; font-weight: bold; margin-bottom: 10px;">Responsibilities</div> <div style="margin-bottom: 5px;">Make appointment</div> <div style="margin-bottom: 5px;">Calculate last visit</div> <div style="margin-bottom: 5px;">Change status</div> <div style="margin-bottom: 5px;">Provide medical history</div> <div style="margin-bottom: 5px;">_____</div> <div style="margin-bottom: 5px;">_____</div> <div style="margin-bottom: 5px;">_____</div> <div style="margin-bottom: 5px;">_____</div>		<div style="text-align: center; font-weight: bold; margin-bottom: 10px;">Collaborators</div> <div style="margin-bottom: 5px;">Appointment</div> <div style="margin-bottom: 5px;">_____</div> <div style="margin-bottom: 5px;">_____</div> <div style="margin-bottom: 5px;">Medical history</div> <div style="margin-bottom: 5px;">_____</div> <div style="margin-bottom: 5px;">_____</div> <div style="margin-bottom: 5px;">_____</div> <div style="margin-bottom: 5px;">_____</div>
Back:		
<div style="font-weight: bold; margin-bottom: 10px;">Attributes:</div> <div style="display: flex; justify-content: space-between;"> <div style="width: 45%;"> <div style="margin-bottom: 5px;">Amount (double)</div> <div style="margin-bottom: 5px;">Insurance carrier (text)</div> <div style="margin-bottom: 5px;">_____</div> <div style="margin-bottom: 5px;">_____</div> <div style="margin-bottom: 5px;">_____</div> <div style="margin-bottom: 5px;">_____</div> </div> <div style="width: 45%;"> <div style="margin-bottom: 5px;">_____</div> <div style="margin-bottom: 5px;">_____</div> <div style="margin-bottom: 5px;">_____</div> <div style="margin-bottom: 5px;">_____</div> </div> </div>		
<div style="font-weight: bold; margin-bottom: 10px;">Relationships:</div> <div style="margin-bottom: 20px;"> Generalization (a-kind-of): Participant <div style="margin-top: 5px;">_____</div> <div style="margin-top: 5px;">_____</div> </div> <div style="margin-bottom: 20px;"> Aggregation (has-parts): <div style="margin-top: 5px;">_____</div> <div style="margin-top: 5px;">_____</div> </div> <div> Other Associations: Appointment, Medical History <div style="margin-top: 5px;">_____</div> <div style="margin-top: 5px;">_____</div> </div>		

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.

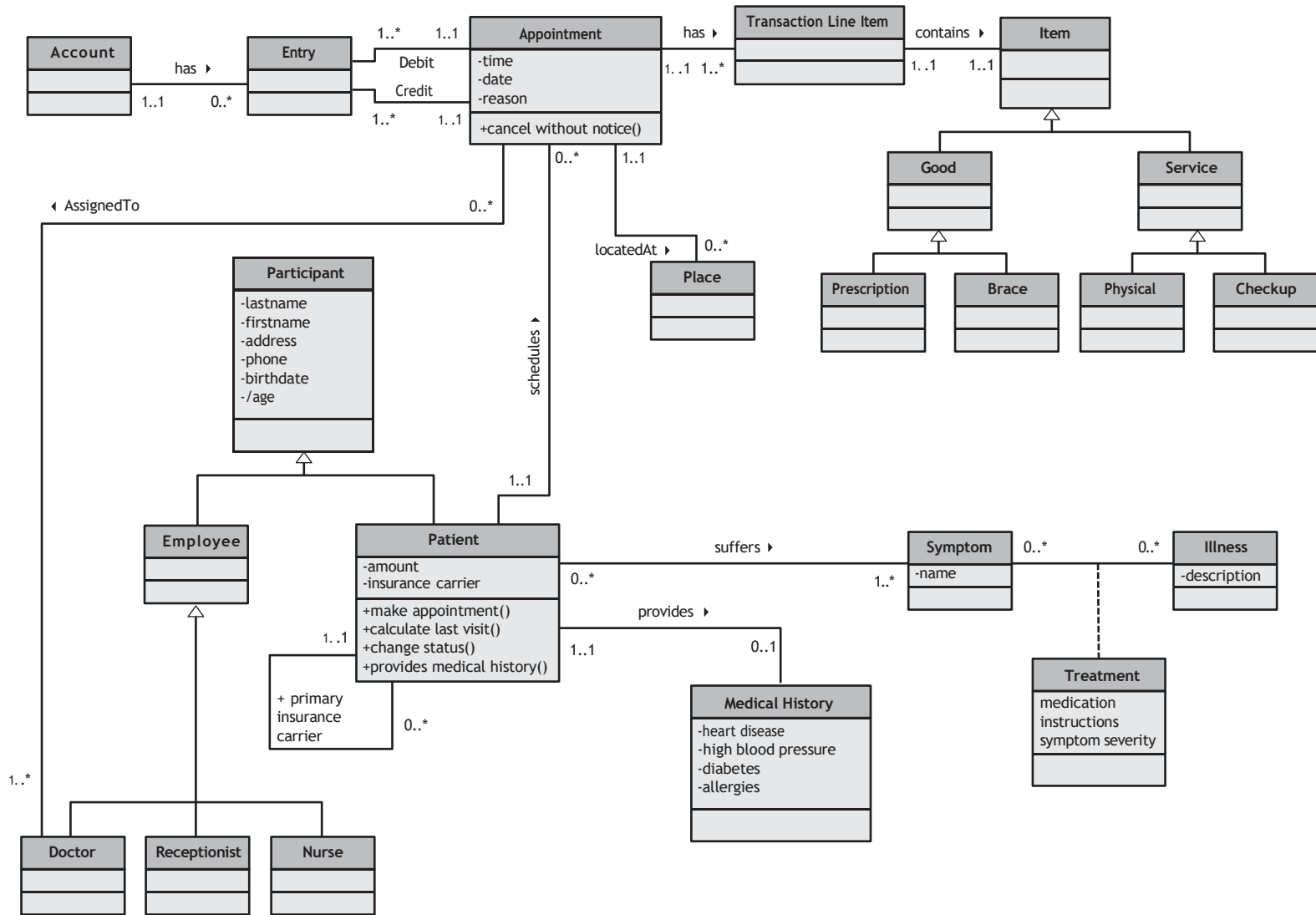


FIGURE 7-4 Appointment Problem Class Diagram (Figure 5-7)

Use-Case Name: Make Old Patient Appt		ID: <u>2</u>	Importance Level: <u>Low</u>
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: <ol style="list-style-type: none"> 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: <p>S-1: New Appointment</p> <ol style="list-style-type: none"> 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. <p>S-2: Cancel Appointment</p> <ol style="list-style-type: none"> 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. <p>S-3: Change Appointment</p> <ol style="list-style-type: none"> 1. The Receptionist performs the S-2: cancel appointment subflow. 2. The Receptionist performs the S-1: new appointment subflow. 			
Alternate/Exceptional Flows: <p>S-1, 2a1: The Receptionist proposes some alternative appointment times based on what is available in the appointment schedule.</p> <p>S-1, 2a2: The Patient chooses one of the proposed times or decides not to make an appointment.</p>			

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

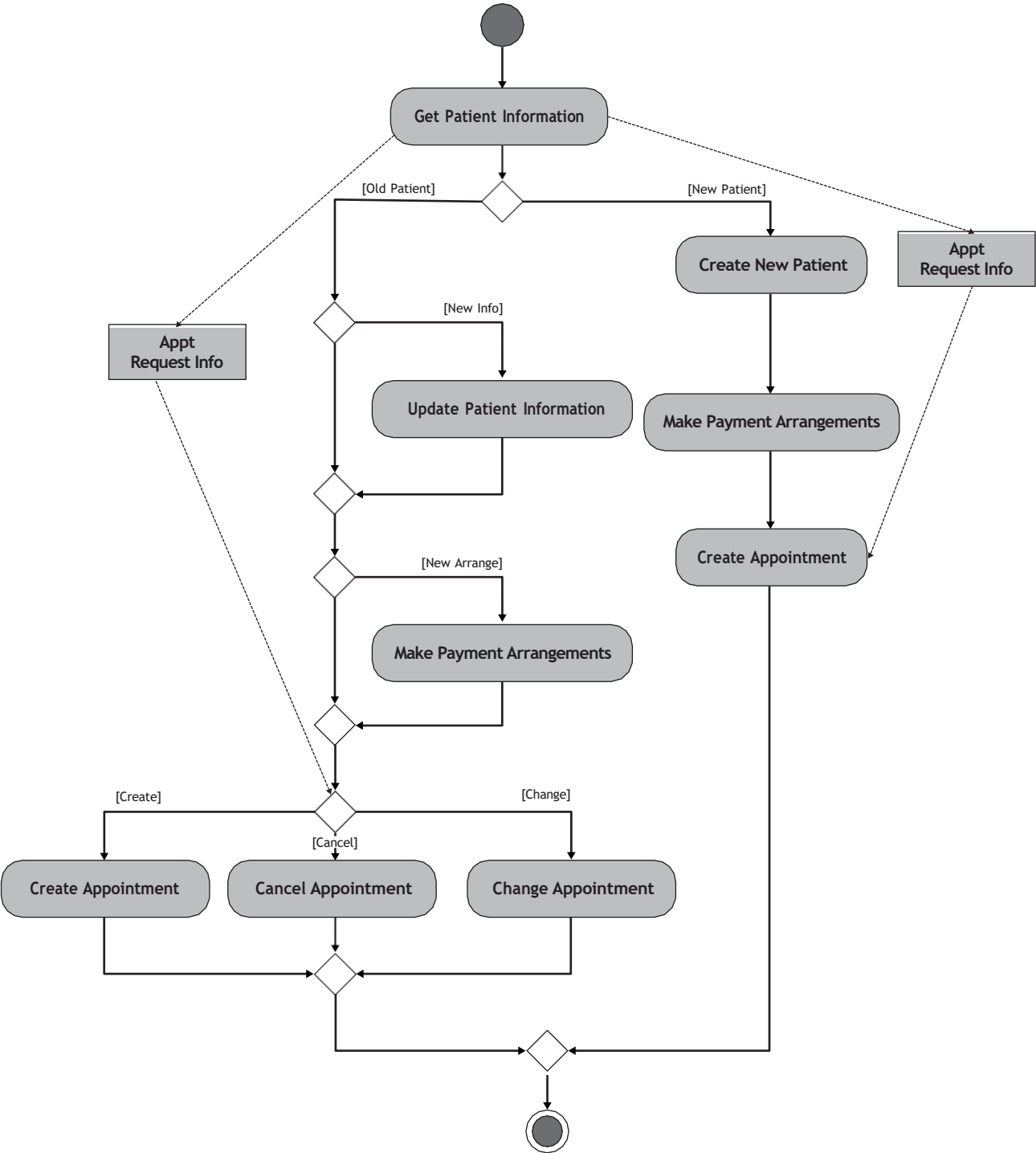


FIGURE 7-6 Activity Diagram for the Manage Appointments Use Case (Figure 4-8)

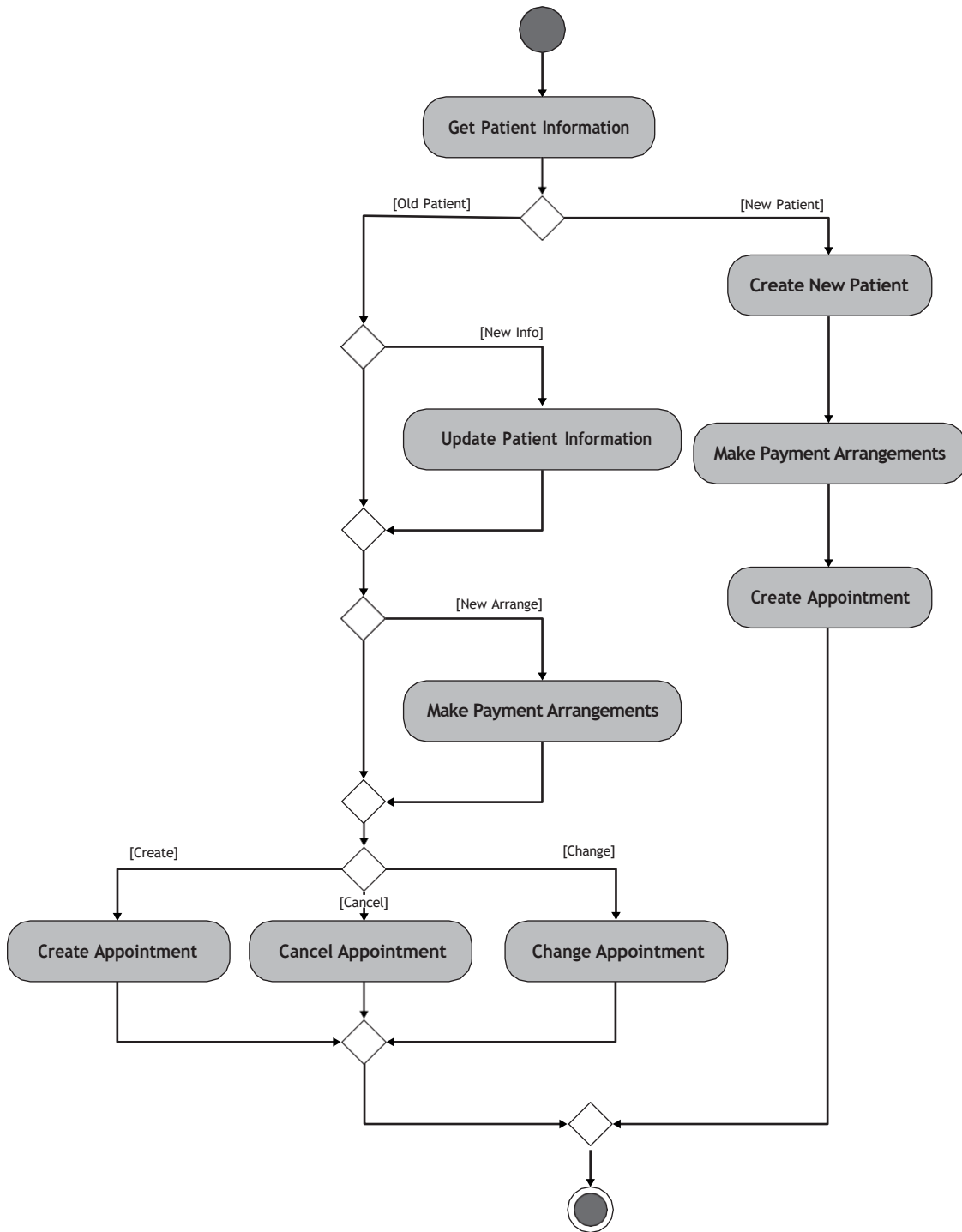


FIGURE 7-7 Corrected Activity Diagram for the Manage Appointments Use Case

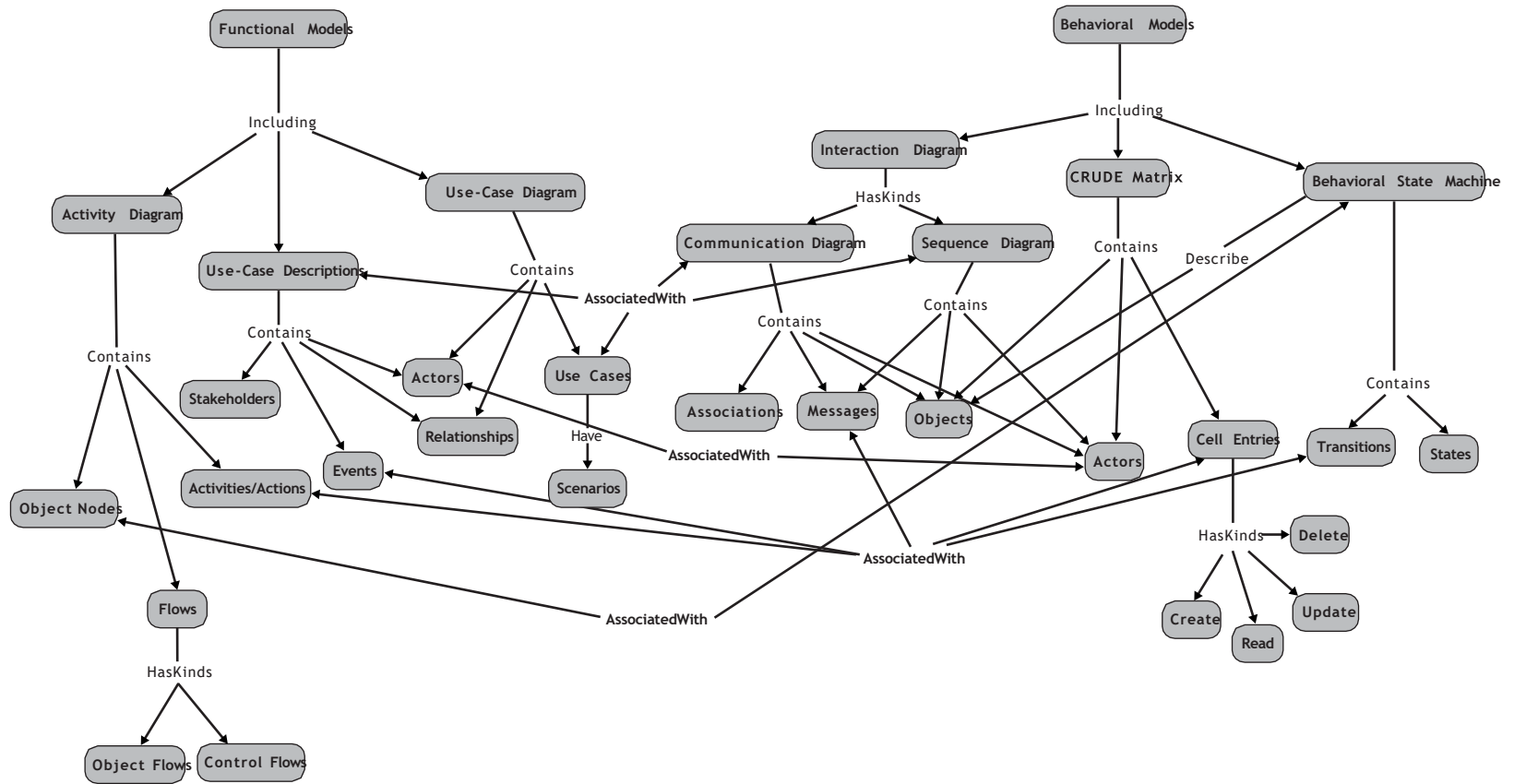
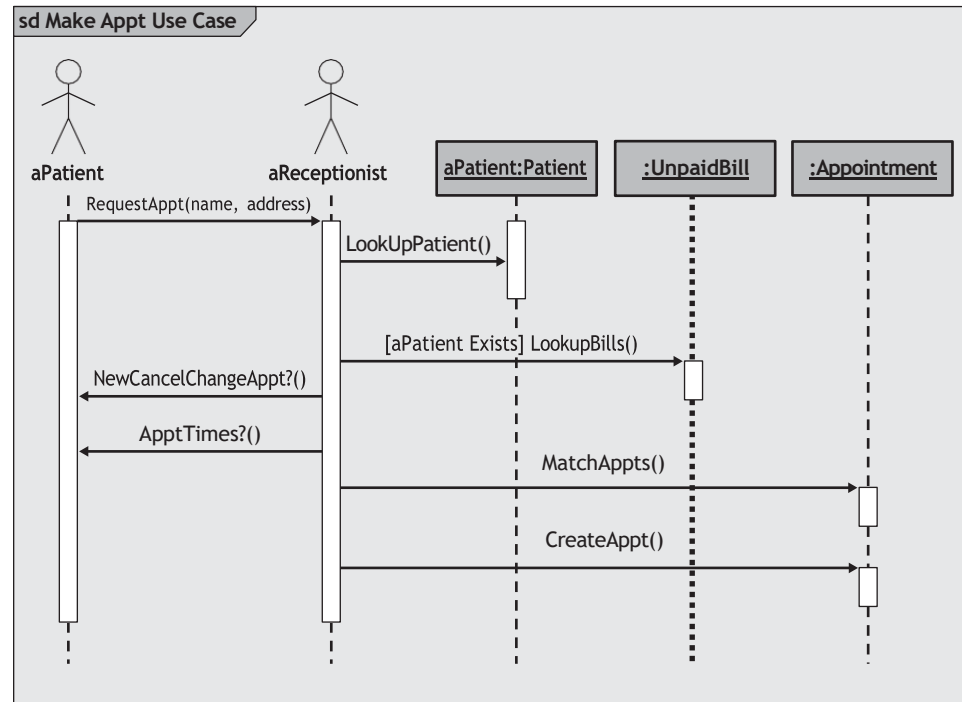


FIGURE 7-8 Relationships between Functional and Behavioral Models

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

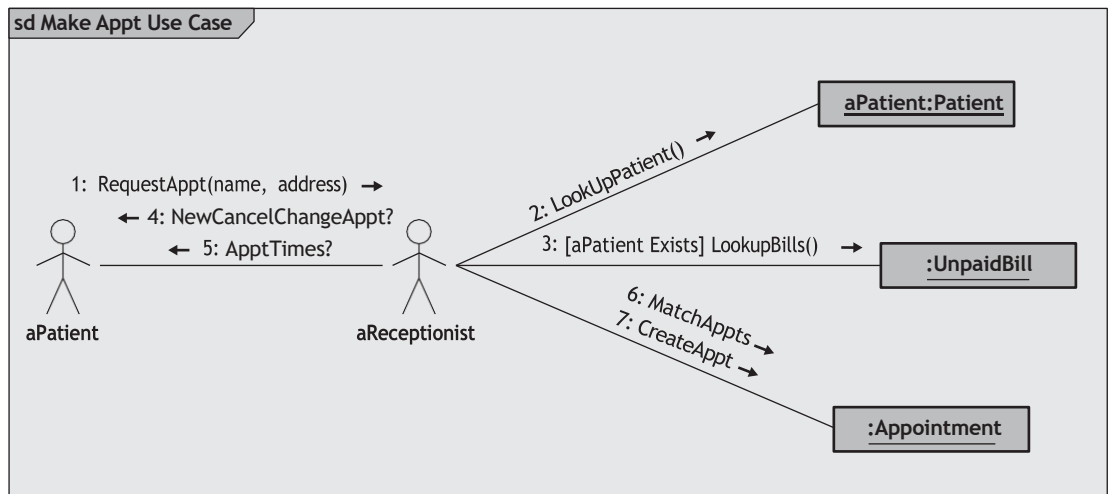


FIGURE 7-10 Communication Diagram for a Scenario of the Make Old Patient Appt Use Case (Figure 6-10)

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

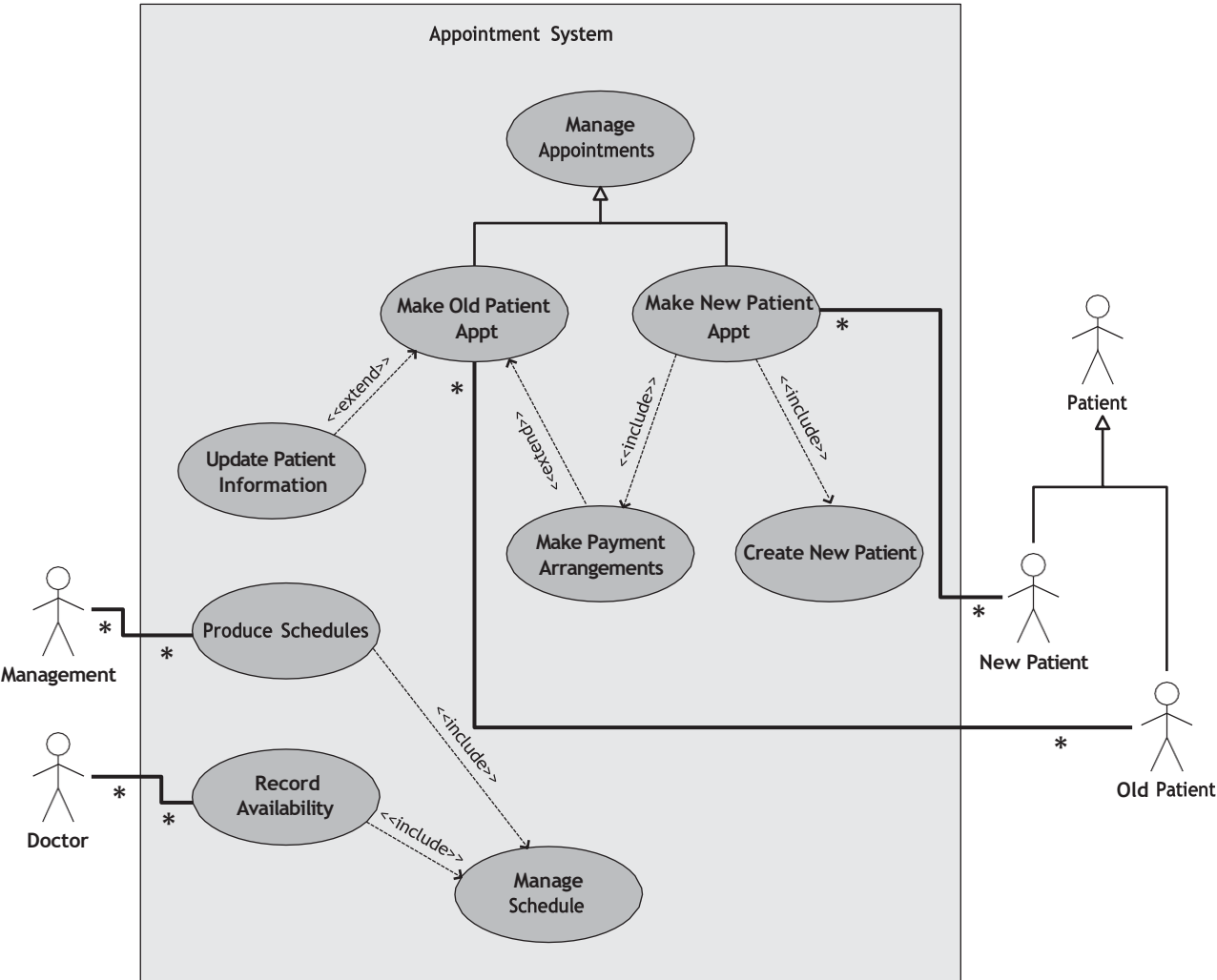


FIGURE 7-11 Modified Use-Case Diagram for the Appointment System (Figure 4-21)

	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)

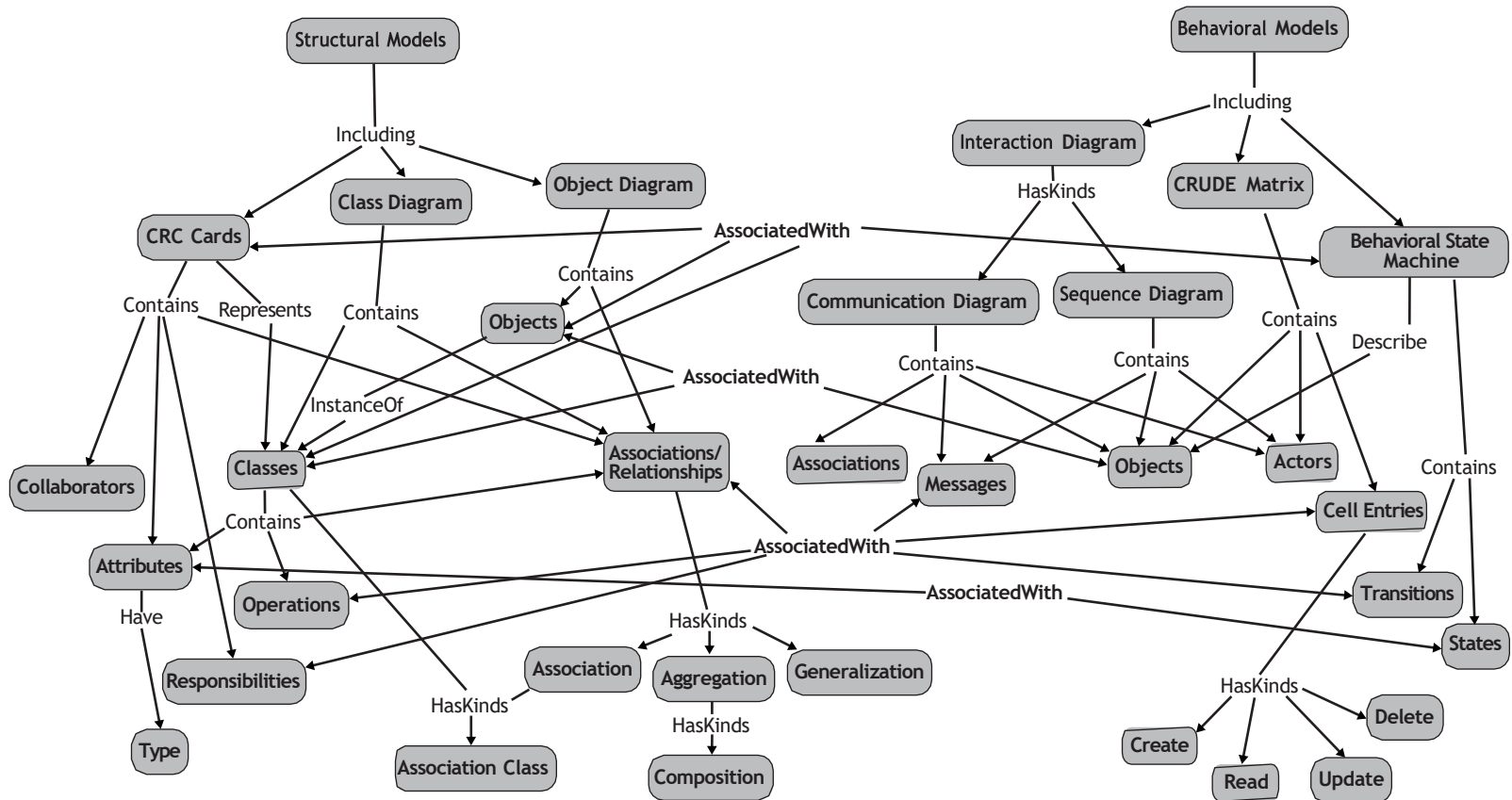


FIGURE 7-13 Relationships between Structural and Behavioral Models

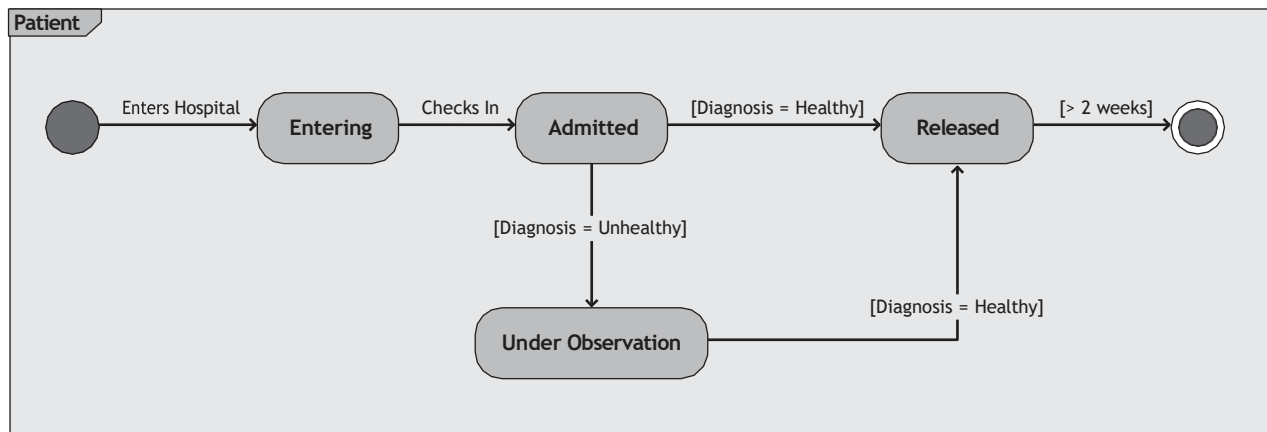


FIGURE 7-14 Behavioral State Machine for Hospital Patient (Figure 6-16)

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

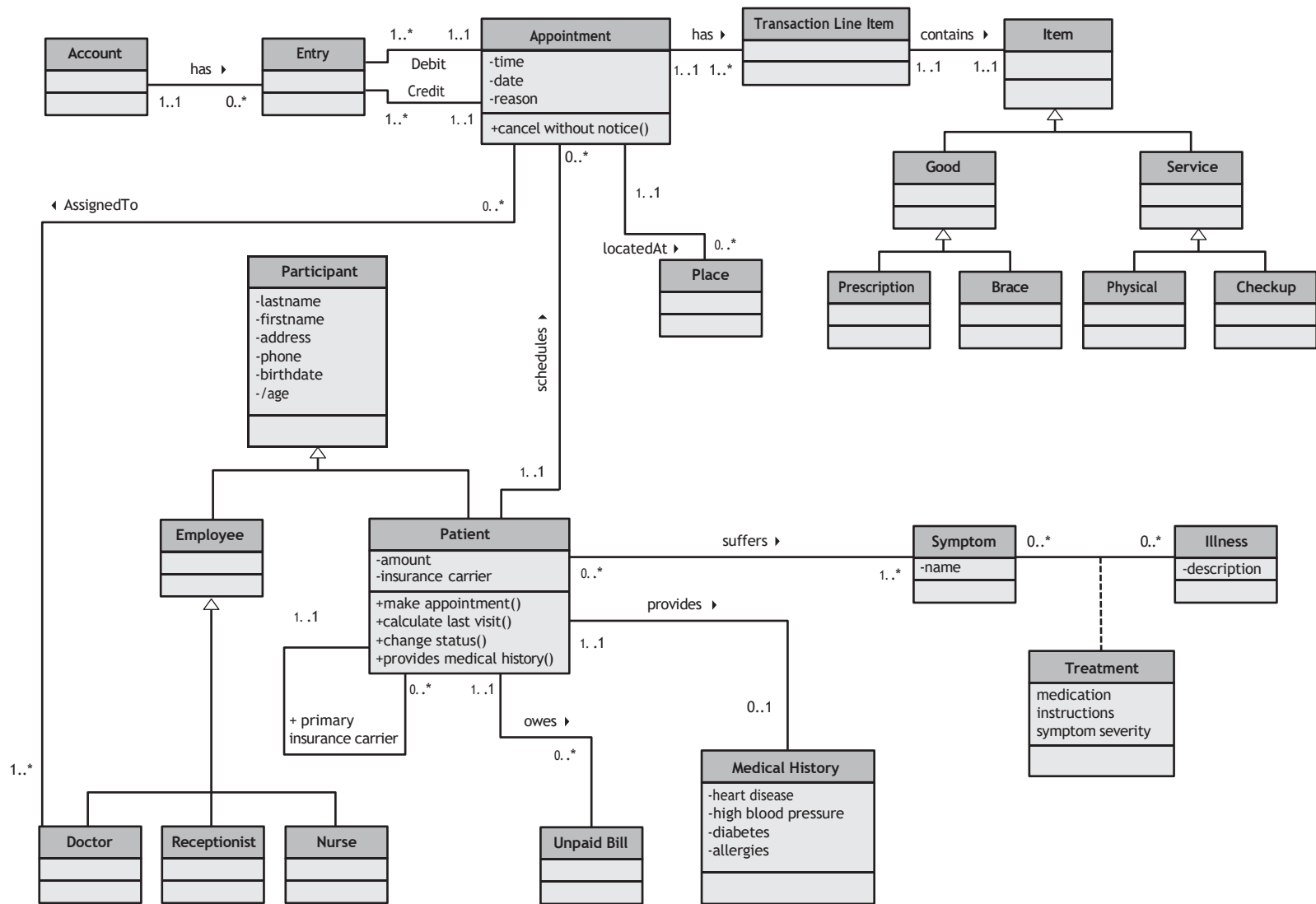


FIGURE 7-15 Corrected Appointment System Class Diagram

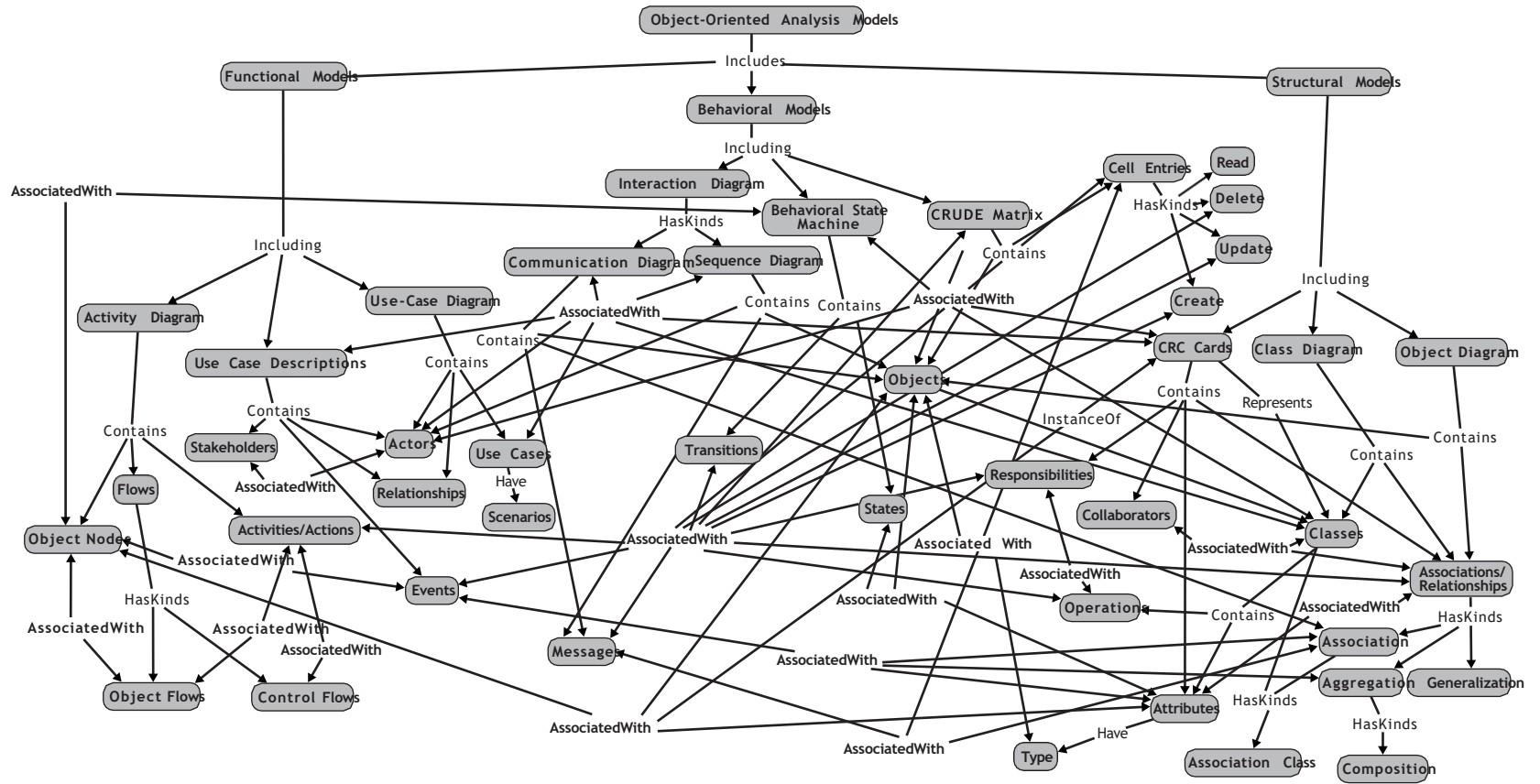


FIGURE 7-16 Interrelationships among Object-Oriented Analysis Models

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.

FIGURE 7-17
Layers and
Sample Classes

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

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. Schneiderman, *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).


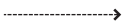
<p>A package:</p> <ul style="list-style-type: none"> ■ Is a logical grouping of UML elements. ■ Is used to simplify UML diagrams by grouping related elements into a single higher-level element. 	
<p>A dependency relationship:</p> <ul style="list-style-type: none"> ■ Represents a dependency between packages: 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 ramifications 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

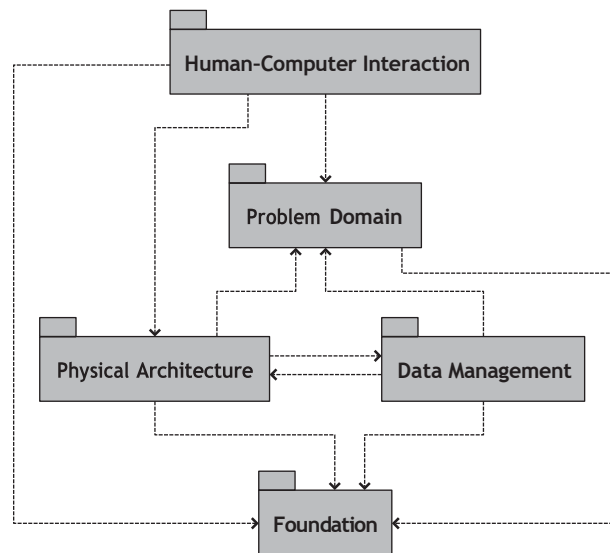


FIGURE 7-19
Package Diagram
of Dependency
Relationships
among Layers

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).

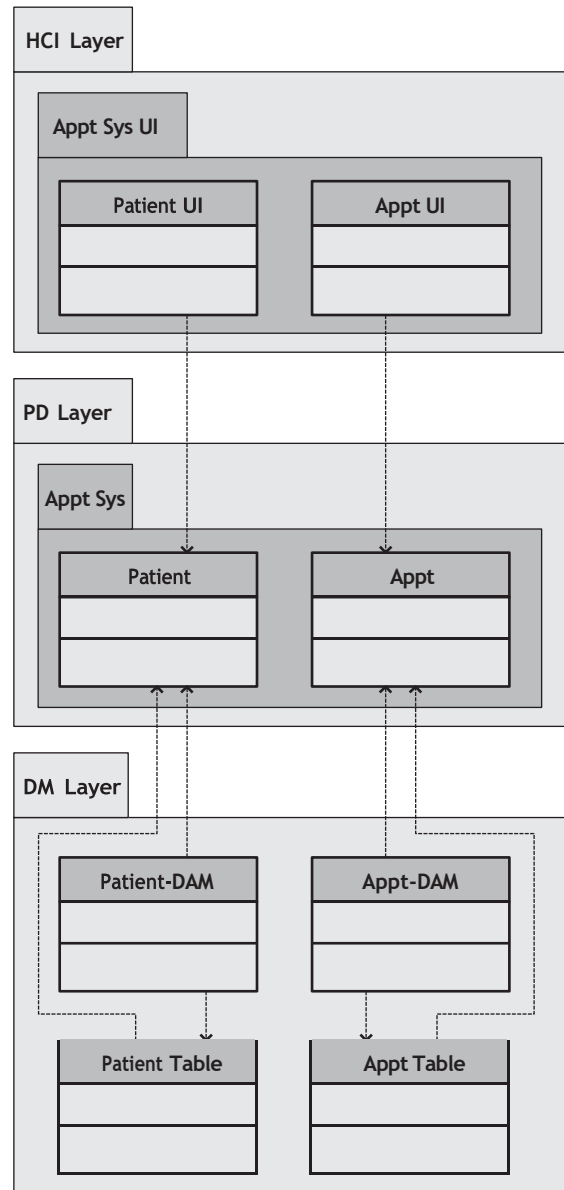


FIGURE 7-20
Partial Package
Diagram of the
Appointment System

- 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

1. Set Context

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.

2. Cluster Classes

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.

3. Create Packages

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.

4. Identify Dependencies

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).

5. Lay Out and Draw Diagram

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).

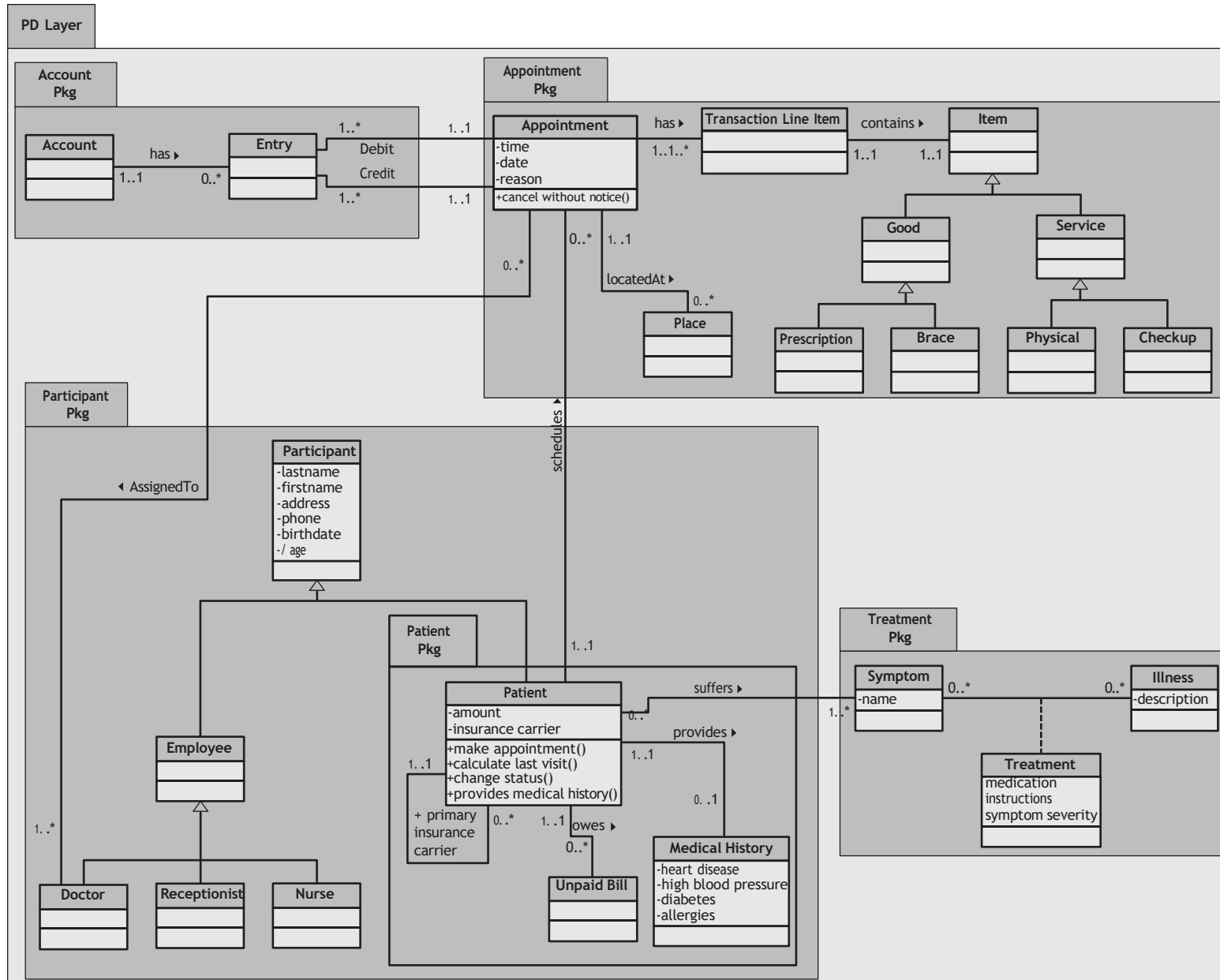


FIGURE 7-21 Package Diagram of the PD Layer of the Appointment Problem

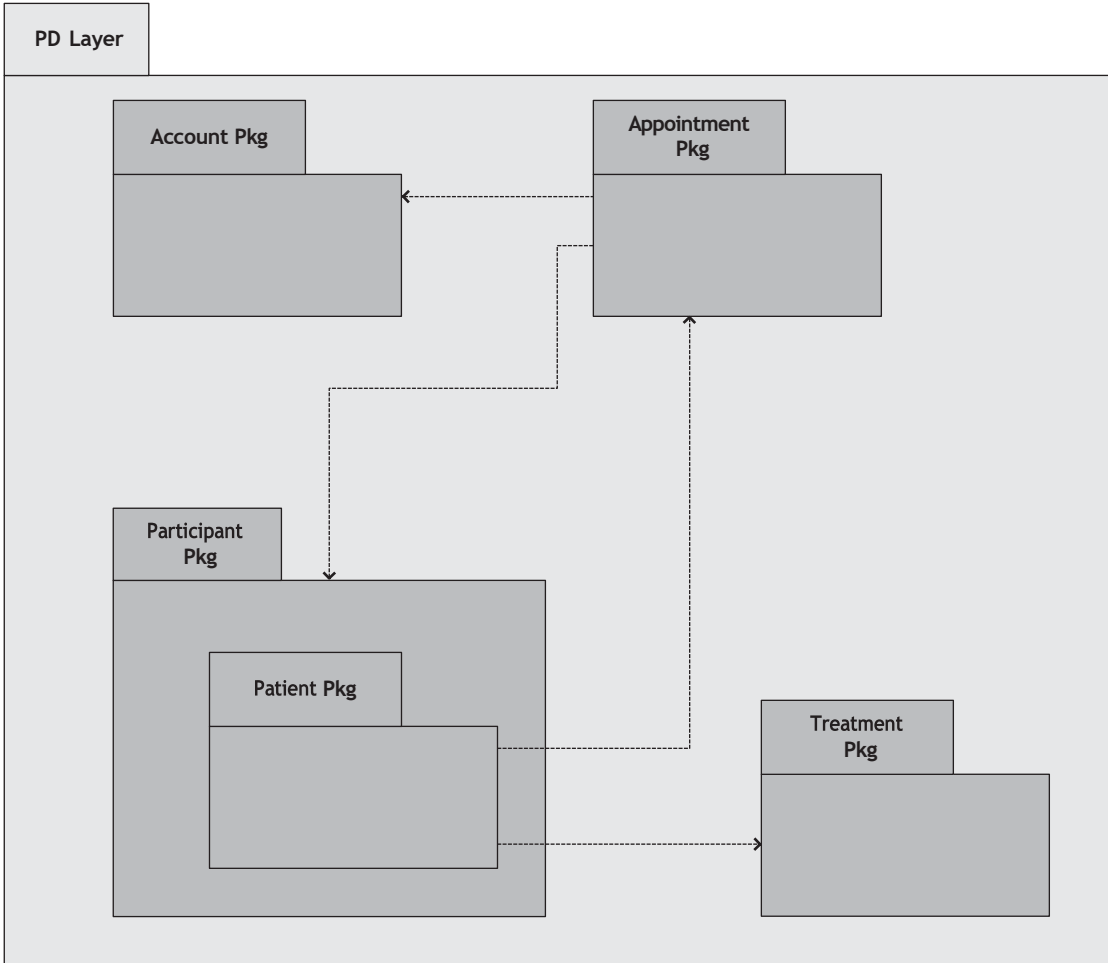


FIGURE 7-22 Overview Package Diagram of the PD Layer for the Appointment System

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).

FIGURE 7-23
Outsourcing
Guidelines

Outsourcing
<ul style="list-style-type: none">• 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.

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 Information	3	75	Supporting Information	3	75	Supporting Information	5	125
Criterion 5	10		3	30		1	10		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	Data management layer	Model	Request for information
Abstract classes	Dependency relationship	Model-View-Controller	(RFI)
Abstraction	Enterprise resource	(MVC)	Request for proposals
Aggregation	systems (ERP)	Module	(RFP)
Alternative matrix	Factoring	Object wrapper	Server
Balancing the models	Fixed-price contract	Outsourcing	Smalltalk
Class	Foundation layer	Package	Systems integration
Client	Generalization	Package diagram	Time-and-arrangements
Collaboration	Has-parts Human–	Packaged software	contract
Concrete classes	computer	Partition	Validation
Contract	interaction layer	Physical architecture	Value-added contract
Controller	Layer	layer	Verification
Custom development	Message	Problem domain layer	View
Customization	Method	Refinement	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.
- 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?

