CHAPTER

6 Requirements Modeling: Scenarios, Information, and Analysis Classes

Key Concepts

At a technical level, software engineering begins with a series of modeling tasks that lead to a specification of requirements and a design representation for the software to be built. The requirements model[1]—actually a set of models—is the first technical representation of a system.

In a seminal book on requirements modeling methods, Tom DeMarco [DeM79] describes the process in this way:

> Looking back over the recognized problems and failings of the analysis phase, I suggest that we need to make the following additions to our set of analysis phase goals. The products of analysis must be highly maintainable. This applies particularly to the

Quick Look

What is it? The written word is a wonderful vehicle for communication, but it is not necessarily the best way to represent the requirements for computer software. Requirements modeling uses a combination of text and diagrammatic forms to depict requirements in a way that is relatively easy to understand, and more important, straightforward to review for correctness, completeness, and consistency.

Who does it? A software engineer (sometimes called an "analyst") builds the model using requirements elicited from the customer.

Why is it important? To validate software requirements, you need to examine them from a number of different points of view. In this chapter you'll consider requirements modeling from three different perspectives: scenario-based models, data (information) models, and class-based models. Each represents requirements in a different "dimension," thereby increasing the probability that errors will be found, that inconsistency will surface, and that omissions will be uncovered.

What are the steps? Scenario-based modeling represents the system from the user's point of view. Data modeling represents the information space and depicts the data objects that the software will manipulate and the relationships among them. Class-based modeling defines objects, attributes, and relationships. Once preliminary models are created, they are refined and analyzed to assess their clarity, completeness, and consistency. In Chapter 7, we extend the modeling dimensions noted here with additional representations, providing a more robust view of requirements.

What is the work product? A wide array of text-based and diagrammatic forms may be chosen for the requirements model. Each of these representations provides a view of one or more of the model elements.

How do I ensure that I've done it right? Requirements modeling work products must be reviewed for correctness, completeness, and consistency. They must reflect the needs of all stakeholders and establish a foundation from which design can be conducted.

1 In past editions of this book, I used the term *analysis model,* rather than *requirements model.* In this edition, I've decided to use both phrases to represent the modeling activity that defines various aspects of the problem to be solved. *Analysis* is the action that occurs as *requirements* are derived.

Target Document [software requirements specification]. Problems of size must be dealt with using an effective method of partitioning. The Victorian novel specification is out. Graphics have to be used whenever possible. We have to differentiate between logical [essential] and physical [implementation] considerations. . . . At the very least, we need. . . . Something to help us partition our requirements and document that partitioning before specification. . . . Some means of keeping track of and evaluating interfaces. . . . New tools to describe logic and policy, something better than narrative text.

Although DeMarco wrote about the attributes of analysis modeling more than a quarter century ago, his comments still apply to modern requirements modeling methods and notation.

6.1 Requirements Analysis

Requirements analysis results in the specification of software's operational characteristics, indicates software's interface with other system elements, and establishes constraints that software must meet. Requirements analysis allows you (regardless of whether you're called a *software engineer,* an *analyst,* or a *modeler*) to elaborate on basic requirements established during the inception, elicitation, and negotiation tasks that are part of requirements engineering (Chapter 5).

Quote:

"Any one 'view' of requirements is insufficient to understand or describe the desired behavior of a complex system."

Alan M. Davis

The requirements modeling action results in one or more of the following types of models:

- *Scenario-based models* of requirements from the point of view of various system "actors"
- *Data models* that depict the information domain for the problem
- *Class-oriented models* that represent object-oriented classes (attributes and operations) and the manner in which classes collaborate to achieve system requirements
- *Flow-oriented models* that represent the functional elements of the system and how they transform data as it moves through the system
- *Behavioral models* that depict how the software behaves as a consequence of external "events"

KEY POINT

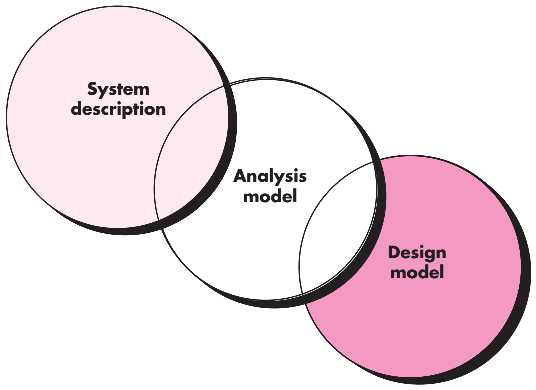
The analysis model and requirements specification provide a means for assessing quality once the software is built.

These models provide a software designer with information that can be translated to architectural, interface, and component-level designs. Finally, the requirements model (and the software requirements specification) provides the developer and the customer with the means to assess quality once software is built.

In this chapter, I focus on *scenario-based modeling*—a technique that is growing increasingly popular throughout the software engineering community; *data modeling*—a more specialized technique that is particularly appropriate when an application must create or manipulate a complex information space; and *class*

FIGURE 6.1

The requirements model as a bridge between the system description and the design model



modeling—a representation of the object-oriented classes and the resultant collaborations that allow a system to function. Flow-oriented models, behavioral models, pattern-based modeling, and WebApp models are discussed in Chapter 7.

6.1.1 Overall Objectives and Philosophy

Quote:

"Requirements are not architecture. Requirements are not design, nor are they the user interface. Requirements are need."

Andrew Hunt and David Thomas

KEY POINT

The analysis model should describe what the customer wants, establish a basis for design, and establish a target for validation.

Throughout requirements modeling, your primary focus is on *what,* not *how.* What user interaction occurs in a particular circumstance, what objects does the system manipulate, what functions must the system perform, what behaviors does the system exhibit, what interfaces are defined, and what constraints apply?[2]

In earlier chapters, I noted that complete specification of requirements may not be possible at this stage. The customer may be unsure of precisely what is required for certain aspects of the system. The developer may be unsure that a specific approach will properly accomplish function and performance. These realities mitigate in favor of an iterative approach to requirements analysis and modeling. The analyst should model what is known and use that model as the basis for design of the software increment.[3]

The requirements model must achieve three primary objectives: (1) to describe what the customer requires, (2) to establish a basis for the creation of a software design, and (3) to define a set of requirements that can be validated once the software is built. The analysis model bridges the gap between a system-level description that describes overall system or business functionality as it is achieved by applying software, hardware, data, human, and other system elements and a software design (Chapters 8 through 13) that describes the software's application architecture, user interface, and component-level structure. This relationship is illustrated in Figure 6.1.

2 It should be noted that as customers become more technologically sophisticated, there is a trend toward the specification of *how* as well as *what.* However, the primary focus should remain on *what.*

3 Alternatively, the software team may choose to create a prototype (Chapter 2) in an effort to better understand requirements for the system.

It is important to note that all elements of the requirements model will be directly traceable to parts of the design model. A clear division of analysis and design tasks between these two important modeling activities is not always possible. Some design invariably occurs as part of analysis, and some analysis will be conducted during design.

6.1.2 Analysis Rules of Thumb

Arlow and Neustadt [Arl02] suggest a number of worthwhile rules of thumb that should be followed when creating the analysis model:

? Are there basic guidelines that can help us as we do requirements analysis work?

- *The model should focus on requirements that are visible within the problem or business domain. The level of abstraction should be relatively high.* "Don't get bogged down in details" [Arl02] that try to explain how the system will work.
- *Each element of the requirements model should add to an overall understanding of software requirements and provide insight into the information domain, function, and behavior of the system.*
- *Delay consideration of infrastructure and other nonfunctional models until design.* That is, a database may be required, but the classes necessary to implement it, the functions required to access it, and the behavior that will be exhibited as it is used should be considered only after problem domain analysis has been completed.
- *Minimize coupling throughout the system.* It is important to represent relationships between classes and functions. However, if the level of "interconnectedness" is extremely high, effort should be made to reduce it.
- *Be certain that the requirements model provides value to all stakeholders.* Each constituency has its own use for the model. For example, business stakeholders should use the model to validate requirements; designers should use the model as a basis for design; QA people should use the model to help plan acceptance tests.
- *Keep the model as simple as it can be.* Don't create additional diagrams when they add no new information. Don't use complex notational forms, when a simple list will do.

Quote:

"Problems worthy of attack, prove their worth by hitting back."

Piet Hein

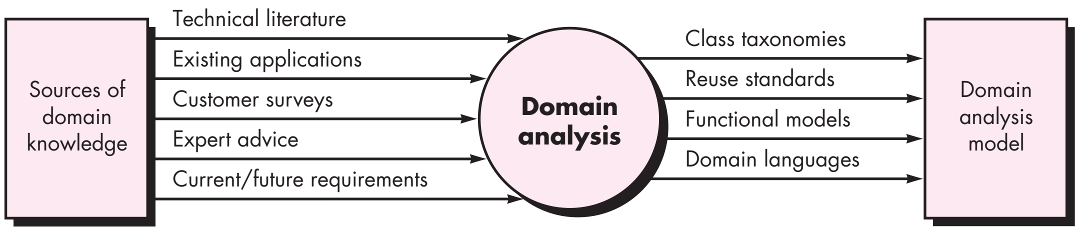
6.1.3 Domain Analysis

WebRef

Many useful resources for domain analysis can be found at **www.iturls.com/English/Software Engineering/SE_mod5.asp**.

In the discussion of requirements engineering (Chapter 5), I noted that analysis patterns often reoccur across many applications within a specific business domain. If these patterns are defined and categorized in a manner that allows you to recognize and apply them to solve common problems, the creation of the analysis model is expedited. More important, the likelihood of applying design patterns and executable software components grows dramatically. This improves time-to-market and reduces development costs.

FIGURE 6.2 Input and output for domain analysis



But how are analysis patterns and classes recognized in the first place? Who defines them, categorizes them, and readies them for use on subsequent projects? The answers to these questions lie in *domain analysis.* Firesmith [Fir93] describes domain analysis in the following way:

KEY POINT
Domain analysis doesn't look at a specific application, but rather at the domain in which the application resides. The intent is to identify common problem solving elements that are applicable to all applications within the domain.

> Software domain analysis is the identification, analysis, and specification of common requirements from a specific application domain, typically for reuse on multiple projects within that application domain. . . . [Object-oriented domain analysis is] the identification, analysis, and specification of common, reusable capabilities within a specific application domain, in terms of common objects, classes, subassemblies, and frameworks.

The "specific application domain" can range from avionics to banking, from multimedia video games to software embedded within medical devices. The goal of domain analysis is straightforward: to find or create those analysis classes and/or analysis patterns that are broadly applicable so that they may be reused.[4]

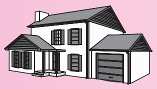
Using terminology that was introduced earlier in this book, domain analysis may be viewed as an umbrella activity for the software process. By this I mean that domain analysis is an ongoing software engineering activity that is not connected to any one software project. In a way, the role of a domain analyst is similar to the role of a master toolsmith in a heavy manufacturing environment. The job of the toolsmith is to design and build tools that may be used by many people doing similar but not necessarily the same jobs. The role of the domain analyst[5] is to discover and define analysis patterns, analysis classes, and related information that may be used by many people working on similar but not necessarily the same applications.

Figure 6.2 [Ara89] illustrates key inputs and outputs for the domain analysis process. Sources of domain knowledge are surveyed in an attempt to identify objects that can be reused across the domain.

4 A complementary view of domain analysis "involves modeling the domain so that software engineers and other stakeholders can better learn about it . . . not all domain classes necessarily result in the development of reusable classes . . ." [Let03a].

5 Do not make the assumption that because a domain analyst is at work, a software engineer need not understand the application domain. Every member of a software team should have some understanding of the domain in which the software is to be placed.

SAFEHOME



Domain Analysis

The scene: Doug Miller's office, after a meeting with marketing.

The players: Doug Miller, software engineering manager, and Vinod Raman, a member of the software engineering team.

The conversation:

Doug: I need you for a special project, Vinod. I'm going to pull you out of the requirements gathering meetings.

Vinod (frowning): Too bad. That format actually works . . . I was getting something out of it. What's up?

Doug: Jamie and Ed will cover for you. Anyway, marketing insists that we deliver the Internet capability along with the home security function in the first release of *SafeHome.* We're under the gun on this . . . not enough time or people, so we've got to solve both problems—the PC interface and the Web interface—at once.

Vinod (looking confused): I didn't know the plan was set . . . we're not even finished with requirements gathering.

Doug (a wan smile): I know, but the time lines are so short that I decided to begin strategizing with marketing right now . . . anyhow, we'll revisit any tentative plan once we have the info from all of the requirements gathering meetings.

Vinod: Okay, what's up? What do you want me to do?

Doug: Do you know what "domain analysis" is?

Vinod: Sort of. You look for similar patterns in Apps that do the same kinds of things as the App you're building. If possible, you then steal the patterns and reuse them in your work.

Doug: Not sure I like the word *steal,* but basically you have it right. What I'd like you to do is to begin researching existing user interfaces for systems that control something like *SafeHome*. I want you to propose a set of patterns and analysis classes that can be common to both the PC-based interface that'll sit in the house and the browser-based interface that is accessible via the Internet.

Vinod: We can save time by making them the same . . . why don't we just do that?

Doug: Ah . . . it's nice to have people who think like you do. That's the whole point—we can save time and effort if both interfaces are nearly identical, implemented with the same code, blah, blah, that marketing insists on.

Vinod: So you want, what—classes, analysis patterns, design patterns?

Doug: All of 'em. Nothing formal at this point. I just want to get a head start on our internal analysis and design work.

Vinod: I'll go to our class library and see what we've got. I'll also use a patterns template I saw in a book I was reading a few months back.

Doug: Good. Go to work.

Quote:

"... analysis is frustrating, full of complex interpersonal relationships, indefinite, and difficult. In a word, it is fascinating. Once you're hooked, the old easy pleasures of system building are never again enough to satisfy you."

Tom DeMarco

6.1.4 Requirements Modeling Approaches

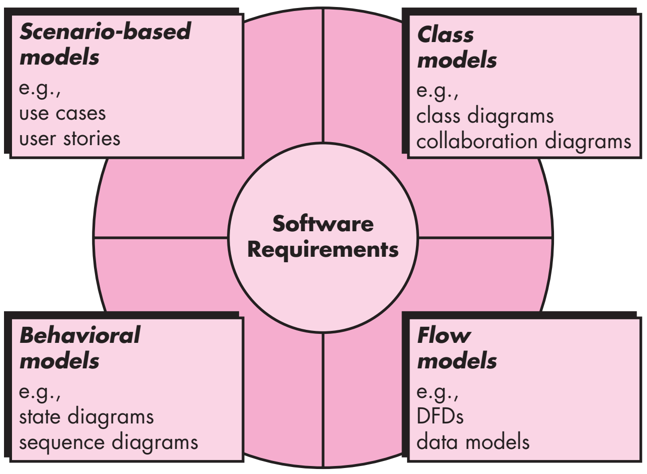
One view of requirements modeling, called *structured analysis,* considers data and the processes that transform the data as separate entities. Data objects are modeled in a way that defines their attributes and relationships. Processes that manipulate data objects are modeled in a manner that shows how they transform data as data objects flow through the system.

A second approach to analysis modeling, called *object-oriented analysis,* focuses on the definition of classes and the manner in which they collaborate with one another to effect customer requirements. UML and the Unified Process (Chapter 2) are predominantly object oriented.

Although the requirements model proposed in this book combines features of both approaches, software teams often choose one approach and exclude all representations from the other. The question is not which is best, but rather, what

FIGURE 6.3

Elements of the analysis model



? What different points of view can be used to describe the requirements model?

Quote:

"Why should we build models? Why not just build the system itself? The answer is that we can construct models in such a way as to highlight, or emphasize, certain critical features of a system, while simultaneously de-emphasizing other aspects of the system."

Ed Yourdon

combination of representations will provide stakeholders with the best model of software requirements and the most effective bridge to software design.

Each element of the requirements model (Figure 6.3) presents the problem from a different point of view. Scenario-based elements depict how the user interacts with the system and the specific sequence of activities that occur as the software is used. Class-based elements model the objects that the system will manipulate, the operations that will be applied to the objects to effect the manipulation, relationships (some hierarchical) between the objects, and the collaborations that occur between the classes that are defined. Behavioral elements depict how external events change the state of the system or the classes that reside within it. Finally, flow-oriented elements represent the system as an information transform, depicting how data objects are transformed as they flow through various system functions.

Analysis modeling leads to the derivation of each of these modeling elements. However, the specific content of each element (i.e., the diagrams that are used to construct the element and the model) may differ from project to project. As we have noted a number of times in this book, the software team must work to keep it simple. Only those modeling elements that add value to the model should be used.

6.2 Scenario-Based Modeling

Although the success of a computer-based system or product is measured in many ways, user satisfaction resides at the top of the list. If you understand how end users (and other actors) want to interact with a system, your software team will be better able to properly characterize requirements and build meaningful analysis and design models. Hence, requirements modeling with UML[6] begins with the creation of scenarios in the form of use cases, activity diagrams, and swimlane diagrams.

6.2.1 Creating a Preliminary Use Case

Quote:

"[Use cases] are simply an aid to defining what exists outside the system (actors) and what should be performed by the system (use cases)."

Ivar Jacobson

Alistair Cockburn characterizes a use case as a "contract for behavior" [Coc01b]. As we discussed in Chapter 5, the "contract" defines the way in which an actor[7] uses a computer-based system to accomplish some goal. In essence, a use case captures the interactions that occur between producers and consumers of information and the system itself. In this section, I examine how use cases are developed as part of the requirements modeling activity.[8]

In Chapter 5, I noted that a use case describes a specific usage scenario in straightforward language from the point of view of a defined actor. But how do you know (1) what to write about, (2) how much to write about it, (3) how detailed to make your description, and (4) how to organize the description? These are the questions that must be answered if use cases are to provide value as a requirements modeling tool.

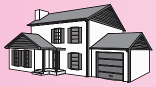
ADVICE

In some situations, use cases become the dominant requirements engineering mechanism. However, this does not mean that you should discard other modeling methods when they are appropriate.

What to write about? The first two requirements engineering tasks—inception and elicitation—provide you with the information you'll need to begin writing use cases. Requirements gathering meetings, QFD, and other requirements engineering mechanisms are used to identify stakeholders, define the scope of the problem, specify overall operational goals, establish priorities, outline all known functional requirements, and describe the things (objects) that will be manipulated by the system.

To begin developing a set of use cases, list the functions or activities performed by a specific actor. You can obtain these from a list of required system functions, through conversations with stakeholders, or by an evaluation of activity diagrams (Section 6.3.1) developed as part of requirements modeling.

SAFEHOME



Developing Another Preliminary User Scenario

The scene: A meeting room, during the second requirements gathering meeting.

The players: Jamie Lazar, software team member; Ed Robbins, software team member; Doug Miller, software engineering manager; three members of marketing; a product engineering representative; and a facilitator.

The conversation:

Facilitator: It's time that we begin talking about the *SafeHome* surveillance function. Let's develop a user scenario for access to the surveillance function.

Jamie: Who plays the role of the actor on this?

6 UML will be used as the modeling notation throughout this book. Appendix 1 provides a brief tutorial for those readers who may be unfamiliar with basic UML notation.

7 An actor is not a specific person, but rather a role that a person (or a device) plays within a specific context. An actor "calls on the system to deliver one of its services" [Coc01b].

8 Use cases are a particularly important part of analysis modeling for user interfaces. Interface analysis is discussed in detail in Chapter 11.

Facilitator: I think Meredith (a marketing person) has been working on that functionality. Why don't you play the role?

Meredith: You want to do it the same way we did it last time, right?

Facilitator: Right . . . same way.

Meredith: Well, obviously the reason for surveillance is to allow the homeowner to check out the house while he or she is away, to record and play back video that is captured . . . that sort of thing.

Ed: Will we use compression to store the video?

Facilitator: Good question, Ed, but let's postpone implementation issues for now. Meredith?

Meredith: Okay, so basically there are two parts to the surveillance function . . . the first configures the system including laying out a floor plan—we have to have tools to help the homeowner do this—and the second part is the actual surveillance function itself. Since the layout is part of the configuration activity, I'll focus on the surveillance function.

Facilitator (smiling): Took the words right out of my mouth.

Meredith: Um . . . I want to gain access to the surveillance function either via the PC or via the Internet. My feeling is that the Internet access would be more frequently used. Anyway, I want to be able to display camera views on a PC and control pan and zoom for a specific camera. I specify the camera by selecting it from the house floor plan. I want to selectively record camera output and replay camera output. I also want to be able to block access to one or more cameras with a specific password. I also want the option of seeing small windows that show views from all cameras and then be able to pick the one I want enlarged.

Jamie: Those are called thumbnail views.

Meredith: Okay, then I want thumbnail views of all the cameras. I also want the interface for the surveillance function to have the same look and feel as all other *SafeHome* interfaces. I want it to be intuitive, meaning I don't want to have to read a manual to use it.

Facilitator: Good job. Now, let's go into this function in a bit more detail . . .

The *SafeHome* home surveillance function (subsystem) discussed in the sidebar identifies the following functions (an abbreviated list) that are performed by the **homeowner** actor:

- Select camera to view.
- Request thumbnails from all cameras.
- Display camera views in a PC window.
- Control pan and zoom for a specific camera.
- Selectively record camera output.
- Replay camera output.
- Access camera surveillance via the Internet.

As further conversations with the stakeholder (who plays the role of a homeowner) progress, the requirements gathering team develops use cases for each of the functions noted. In general, use cases are written first in an informal narrative fashion. If more formality is required, the same use case is rewritten using a structured format similar to the one proposed in Chapter 5 and reproduced later in this section as a sidebar.

To illustrate, consider the function *access camera surveillance via the Internet—display camera views* **(ACS-DCV).** The stakeholder who takes on the role of the **homeowner** actor might write the following narrative:

Use case: Access camera surveillance via the Internet—display camera views (ACS-DCV)

Actor: homeowner

If I'm at a remote location, I can use any PC with appropriate browser software to log on to the *SafeHome Products* website. I enter my user ID and two levels of passwords and once I'm validated, I have access to all functionality for my installed *SafeHome* system. To access a specific camera view, I select "surveillance" from the major function buttons displayed. I then select "pick a camera" and the floor plan of the house is displayed. I then select the camera that I'm interested in. Alternatively, I can look at thumbnail snapshots from all cameras simultaneously by selecting "all cameras" as my viewing choice. Once I choose a camera, I select "view" and a one-frame-per-second view appears in a viewing window that is identified by the camera ID. If I want to switch cameras, I select "pick a camera" and the original viewing window disappears and the floor plan of the house is displayed again. I then select the camera that I'm interested in. A new viewing window appears.

A variation of a narrative use case presents the interaction as an ordered sequence of user actions. Each action is represented as a declarative sentence. Revisiting the **ACS-DCV** function, you would write:

Use case: Access camera surveillance via the Internet—display camera views (ACS-DCV)

Actor: homeowner

1. The homeowner logs onto the *SafeHome Products* website.
2. The homeowner enters his or her user ID.
3. The homeowner enters two passwords (each at least eight characters in length).
4. The system displays all major function buttons.
5. The homeowner selects the "surveillance" from the major function buttons.
6. The homeowner selects "pick a camera."
7. The system displays the floor plan of the house.
8. The homeowner selects a camera icon from the floor plan.
9. The homeowner selects the "view" button.
10. The system displays a viewing window that is identified by the camera ID.
11. The system displays video output within the viewing window at one frame per second.

Quote:

"Use cases can be used in many [software] processes. Our favorite is a process that is iterative and risk driven."

Geri Schneider and Jason Winters

It is important to note that this sequential presentation does not consider any alternative interactions (the narrative is more free-flowing and did represent a few alternatives). Use cases of this type are sometimes referred to as *primary scenarios* [Sch98a].

6.2.2 Refining a Preliminary Use Case

A description of alternative interactions is essential for a complete understanding of the function that is being described by a use case. Therefore, each step in the primary scenario is evaluated by asking the following questions [Sch98a]:

? How do I examine alternative courses of action when I develop a use case?

- *Can the actor take some other action at this point?*
- *Is it possible that the actor will encounter some error condition at this point?* If so, what might it be?
- *Is it possible that the actor will encounter some other behavior at this point (e.g., behavior that is invoked by some event outside the actor's control)?* If so, what might it be?

Answers to these questions result in the creation of a set of *secondary scenarios* that are part of the original use case but represent alternative behavior. For example, consider steps 6 and 7 in the primary scenario presented earlier:

6. The homeowner selects "pick a camera."
7. The system displays the floor plan of the house.

Can the actor take some other action at this point? The answer is "yes." Referring to the free-flowing narrative, the actor may choose to view thumbnail snapshots of all cameras simultaneously. Hence, one secondary scenario might be "View thumbnail snapshots for all cameras."

Is it possible that the actor will encounter some error condition at this point? Any number of error conditions can occur as a computer-based system operates. In this context, we consider only error conditions that are likely as a direct result of the action described in step 6 or step 7. Again the answer to the question is "yes." A floor plan with camera icons may have never been configured. Hence, selecting "pick a camera" results in an error condition: "No floor plan configured for this house."[9] This error condition becomes a secondary scenario.

Is it possible that the actor will encounter some other behavior at this point? Again the answer to the question is "yes." As steps 6 and 7 occur, the system may encounter an alarm condition. This would result in the system displaying a special alarm notification (type, location, system action) and providing the actor with a number of options relevant to the nature of the alarm. Because this secondary scenario can occur at any time for virtually all interactions, it will not become part of the **ACS-DCV** use case. Rather, a separate use case—**Alarm condition encountered**—would be developed and referenced from other use cases as required.

9 In this case, another actor, the **system administrator,** would have to configure the floor plan, install and initialize (e.g., assign an equipment ID) all cameras, and test each camera to be certain that it is accessible via the system and through the floor plan.

Each of the situations described in the preceding paragraphs is characterized as a use-case exception. An *exception* describes a situation (either a failure condition or an alternative chosen by the actor) that causes the system to exhibit somewhat different behavior.

Cockburn [Coc01b] recommends using a "brainstorming" session to derive a reasonably complete set of exceptions for each use case. In addition to the three generic questions suggested earlier in this section, the following issues should also be explored:

- *Are there cases in which some "validation function" occurs during this use case?* This implies that validation function is invoked and a potential error condition might occur.
- *Are there cases in which a supporting function (or actor) will fail to respond appropriately?* For example, a user action awaits a response but the function that is to respond times out.
- *Can poor system performance result in unexpected or improper user actions?* For example, a Web-based interface responds too slowly, resulting in a user making multiple selects on a processing button. These selects queue inappropriately and ultimately generate an error condition.

The list of extensions developed as a consequence of asking and answering these questions should be "rationalized" [Co01b] using the following criteria: an exception should be noted within the use case if the software can detect the condition described and then handle the condition once it has been detected. In some cases, an exception will precipitate the development of another use case (to handle the condition noted).

6.2.3 Writing a Formal Use Case

The informal use cases presented in Section 6.2.1 are sometimes sufficient for requirements modeling. However, when a use case involves a critical activity or describes a complex set of steps with a significant number of exceptions, a more formal approach may be desirable.

The **ACS-DCV** use case shown in the sidebar follows a typical outline for formal use cases. The *goal in context* identifies the overall scope of the use case. The *precondition* describes what is known to be true before the use case is initiated. The *trigger* identifies the event or condition that "gets the use case started" [Coc01b]. The *scenario* lists the specific actions that are required by the actor and the appropriate system responses. *Exceptions* identify the situations uncovered as the preliminary use case is refined (Section 6.2.2). Additional headings may or may not be included and are reasonably self-explanatory.

SAFEHOME

Use Case Template for Surveillance

Use case: Access camera surveillance via the Internet—display camera views (ACS-DCV)

Iteration: 2, last modification: January 14 by V. Raman.

Primary actor: Homeowner.

Goal in context: To view output of camera placed throughout the house from any remote location via the Internet.

Preconditions: System must be fully configured; appropriate user ID and passwords must be obtained.

Trigger: The homeowner decides to take a look inside the house while away.

Scenario:

1. The homeowner logs onto the *SafeHome Products* website.
2. The homeowner enters his or her user ID.
3. The homeowner enters two passwords (each at least eight characters in length).
4. The system displays all major function buttons.
5. The homeowner selects the "surveillance" from the major function buttons.
6. The homeowner selects "pick a camera."
7. The system displays the floor plan of the house.
8. The homeowner selects a camera icon from the floor plan.
9. The homeowner selects the "view" button.
10. The system displays a viewing window that is identified by the camera ID.
11. The system displays video output within the viewing window at one frame per second.

Exceptions:

1. ID or passwords are incorrect or not recognized—see use case **Validate ID and passwords.**
2. Surveillance function not configured for this system—system displays appropriate error message; see use case **Configure surveillance function.**
3. Homeowner selects "View thumbnail snapshots for all camera"—see use case **View thumbnail snapshots for all cameras.**
4. A floor plan is not available or has not been configured—display appropriate error message and see use case **Configure floor plan.**
5. An alarm condition is encountered—see use case **Alarm condition encountered.**

Priority: Moderate priority, to be implemented after basic functions.

When available: Third increment.

Frequency of use: Moderate frequency.

Channel to actor: Via PC-based browser and Internet connection.

Secondary actors: System administrator, cameras.

Channels to secondary actors:

1. System administrator: PC-based system.
2. Cameras: wireless connectivity.

Open issues:

1. What mechanisms protect unauthorized use of this capability by employees of *SafeHome Products*?
2. Is security sufficient? Hacking into this feature would represent a major invasion of privacy.
3. Will system response via the Internet be acceptable given the bandwidth required for camera views?
4. Will we develop a capability to provide video at a higher frames-per-second rate when high-bandwidth connections are available?

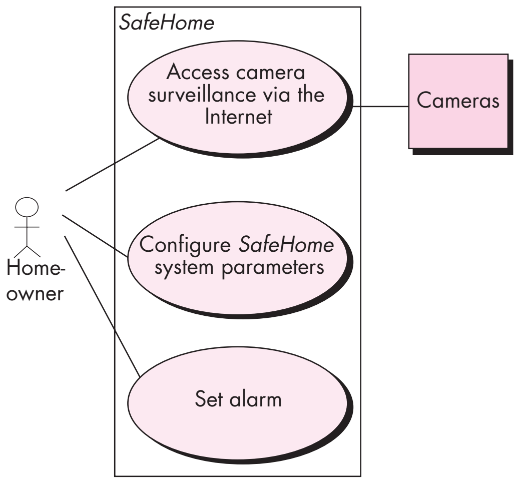
WebRef

When are you finished writing use cases? For a worthwhile discussion of this topic, see **ootips.org/use-cases-done.html**.

In many cases, there is no need to create a graphical representation of a usage scenario. However, diagrammatic representation can facilitate understanding, particularly when the scenario is complex. As we noted earlier in this book, UML does provide use-case diagramming capability. Figure 6.4 depicts a preliminary use-case diagram for the *SafeHome* product. Each use case is represented by an oval. Only the **ACS-DCV** use case has been discussed in this section.

FIGURE 6.4

Preliminary use-case diagram for the *SafeHome* system



SafeHome
Access camera surveillance via the Internet
Configure *SafeHome* system parameters
Set alarm
Cameras
Home-owner

Every modeling notation has limitations, and the use case is no exception. Like any other form of written description, a use case is only as good as its author(s). If the description is unclear, the use case can be misleading or ambiguous. A use case focuses on functional and behavioral requirements and is generally inappropriate for nonfunctional requirements. For situations in which the requirements model must have significant detail and precision (e.g., safety critical systems), a use case may not be sufficient.

However, scenario-based modeling is appropriate for a significant majority of all situations that you will encounter as a software engineer. If developed properly, the use case can provide substantial benefit as a modeling tool.

6.3 UML Models That Supplement the Use Case

There are many requirements modeling situations in which a text-based model—even one as simple as a use case—may not impart information in a clear and concise manner. In such cases, you can choose from a broad array of UML graphical models.

6.3.1 Developing an Activity Diagram

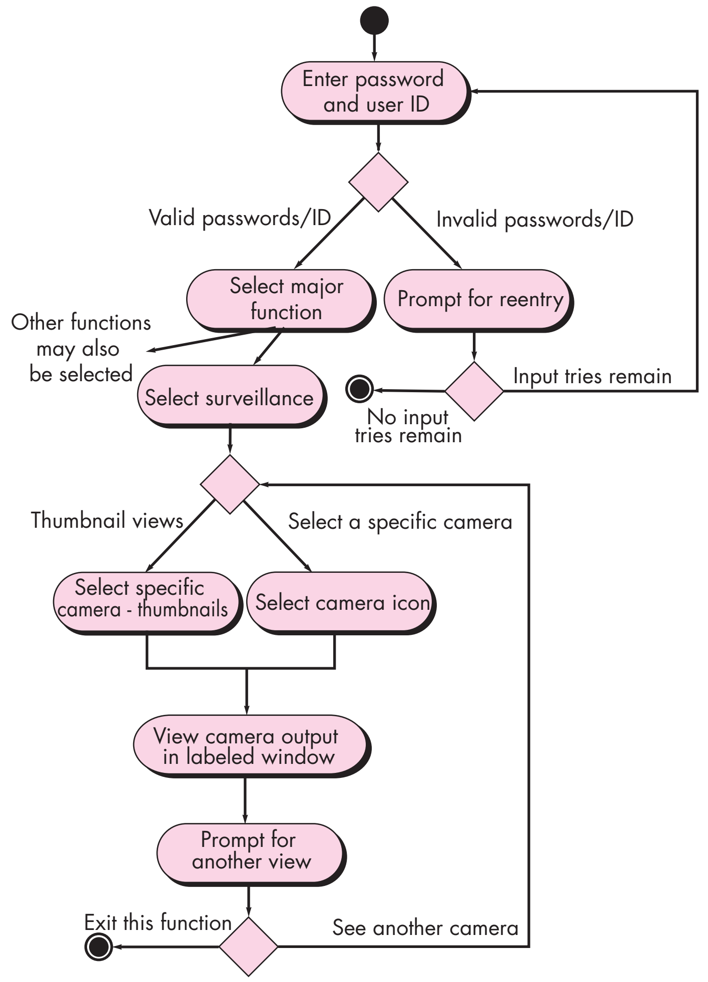
KEY POINT

A UML activity diagram represents the actions and decisions that occur as some function is performed.

The UML activity diagram supplements the use case by providing a graphical representation of the flow of interaction within a specific scenario. Similar to the flowchart, an activity diagram uses rounded rectangles to imply a specific system function, arrows to represent flow through the system, decision diamonds to depict a branching decision (each arrow emanating from the diamond is labeled), and solid horizontal lines to indicate that parallel activities are occurring. An activity diagram for the **ACS-DCV** use case is shown in Figure 6.5. It should be noted that the activity diagram adds additional detail not directly mentioned (but implied) by the use case.

FIGURE 6.5

Activity diagram for Access camera surveillance via the Internet—display camera views function.



For example, a user may only attempt to enter **userID** and **password** a limited number of times. This is represented by a decision diamond below "Prompt for reentry."

6.3.2 Swimlane Diagrams

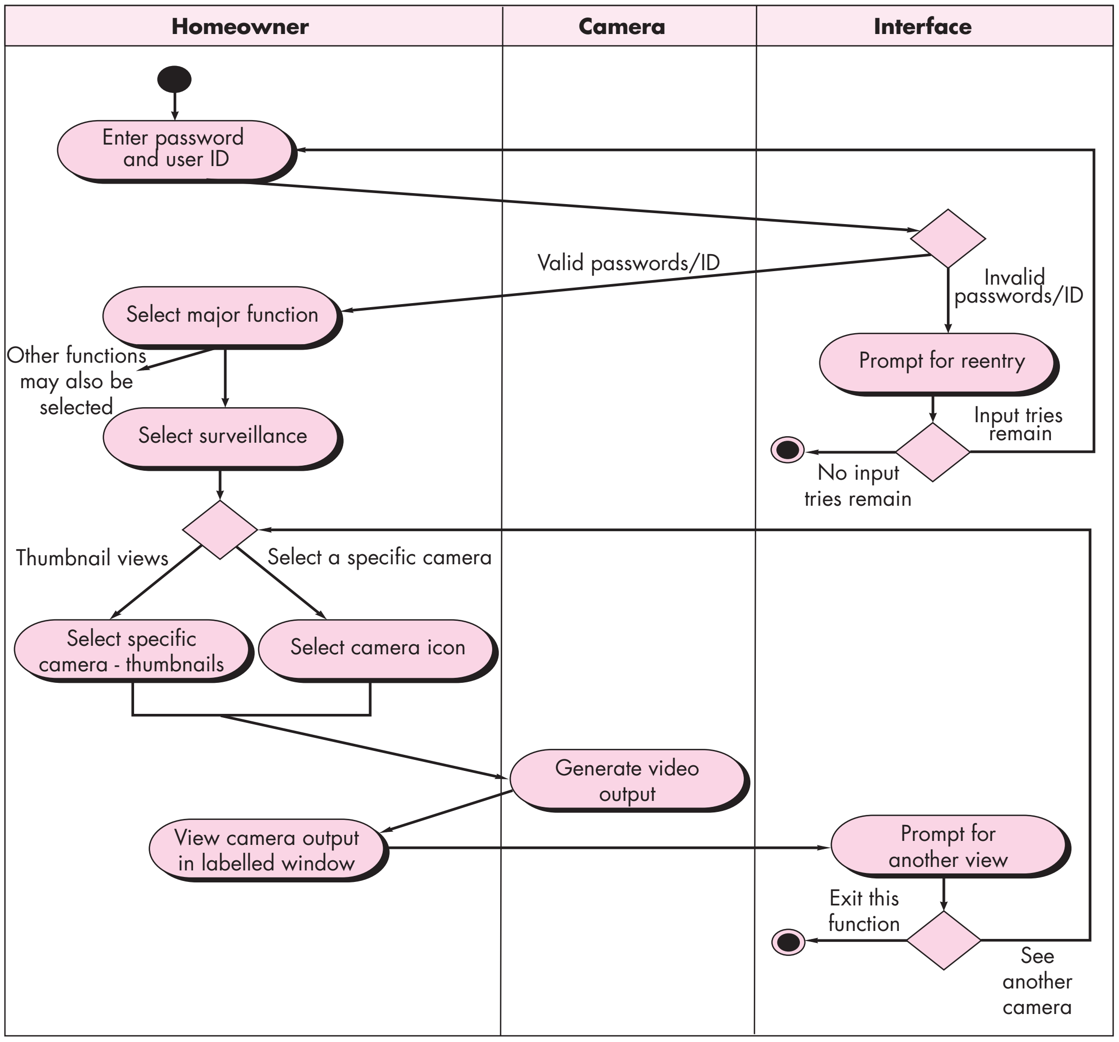
KEY POINT

A UML swimlane diagram represents the flow of actions and decisions and indicates which actors perform each.

The UML *swimlane diagram* is a useful variation of the activity diagram and allows you to represent the flow of activities described by the use case and at the same time indicate which actor (if there are multiple actors involved in a specific use case) or analysis class (discussed later in this chapter) has responsibility for the action described by an activity rectangle. Responsibilities are represented as parallel segments that divide the diagram vertically, like the lanes in a swimming pool.

Three analysis classes—**Homeowner, Camera,** and **Interface**—have direct or indirect responsibilities in the context of the activity diagram represented in Figure 6.5.

FIGURE 6.6 Swimlane diagram for Access camera surveillance via the Internet—display camera views function



Referring to Figure 6.6, the activity diagram is rearranged so that activities associated with a particular analysis class fall inside the swimlane for that class. For example, the **Interface** class represents the user interface as seen by the homeowner. The activity diagram notes two prompts that are the responsibility of the interface—"prompt for reentry" and "prompt for another view." These prompts and the decisions associated with them fall within the **Interface** swimlane. However, arrows lead from that swimlane back to the **Homeowner** swimlane, where homeowner actions occur.

uote:

"A good model guides your thinking, a bad one warps it."

Brian Marick

Use cases, along with the activity and swimlane diagrams, are procedurally oriented. They represent the manner in which various actors invoke specific functions (or other procedural steps) to meet the requirements of the system. But a procedural view of requirements represents only a single dimension of a system. In Section 6.4, I examine the information space and how data requirements can be represented.

6.4 Data Modeling Concepts

WebRef
Useful information on data modeling can be found at **www.datamodel.org**.

If software requirements include the need to create, extend, or interface with a database or if complex data structures must be constructed and manipulated, the software team may choose to create a *data model* as part of overall requirements modeling. A software engineer or analyst defines all data objects that are processed within the system, the relationships between the data objects, and other information that is pertinent to the relationships. The *entity-relationship diagram* (ERD) addresses these issues and represents all data objects that are entered, stored, transformed, and produced within an application.

6.4.1 Data Objects

? How does a data object manifest itself within the context of an application?

A *data object* is a representation of composite information that must be understood by software. By *composite information,* I mean something that has a number of different properties or attributes. Therefore, width (a single value) would not be a valid data object, but **dimensions** (incorporating height, width, and depth) could be defined as an object.

A data object can be an external entity (e.g., anything that produces or consumes information), a thing (e.g., a report or a display), an occurrence (e.g., a telephone call) or event (e.g., an alarm), a role (e.g., salesperson), an organizational unit (e.g., accounting department), a place (e.g., a warehouse), or a structure (e.g., a file). For example, a **person** or a **car** can be viewed as a data object in the sense that either can be defined in terms of a set of attributes. The description of the data object incorporates the data object and all of its attributes.

KEY POINT
A data object is a representation of any composite information that is processed by software.

A data object encapsulates data only—there is no reference within a data object to operations that act on the data.[10] Therefore, the data object can be represented as a table as shown in Figure 6.7. The headings in the table reflect attributes of the object. In this case, a car is defined in terms of **make, model, ID number, body type, color,** and **owner**. The body of the table represents specific instances of the data object. For example, a Chevy Corvette is an instance of the data object **car.**

KEY POINT
Attributes name a data object, describe its characteristics, and in some cases, make reference to another object.

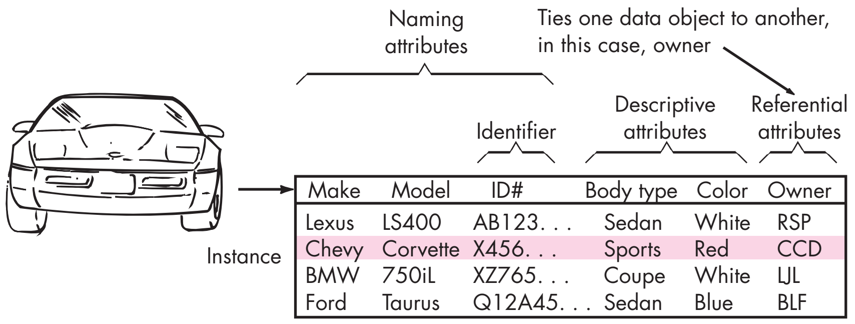
6.4.2 Data Attributes

Data attributes define the properties of a data object and take on one of three different characteristics. They can be used to (1) name an instance of the data object, (2) describe the instance, or (3) make reference to another instance in another table. In addition, one or more of the attributes must be defined as an identifier—that is, the identifier

10 This distinction separates the data object from the class or object defined as part of the object-oriented approach (Appendix 2).

FIGURE 6.7

Tabular representation of data objects



Make	Model	ID#	Body type	Color	Owner
Lexus	LS400	AB123. . .	Sedan	White	RSP
Chevy	Corvette	X456. . .	Sports	Red	CCD
BMW	750iL	XZ765. . .	Coupe	White	LJL
Ford	Taurus	Q12A45. . .	Sedan	Blue	BLF

attribute becomes a "key" when we want to find an instance of the data object. In some cases, values for the identifier(s) are unique, although this is not a requirement. Referring to the data object **car,** a reasonable identifier might be the **ID number.**

WebRef

A concept called "normalization" is important to those who intend to do thorough data modeling. A useful introduction can be found at **www.datamodel.org**.

The set of attributes that is appropriate for a given data object is determined through an understanding of the problem context. The attributes for **car** might serve well for an application that would be used by a department of motor vehicles, but these attributes would be useless for an automobile company that needs manufacturing control software. In the latter case, the attributes for **car** might also include **ID number, body type,** and **color,** but many additional attributes (e.g., **interior code, drive train type, trim package designator, transmission type**) would have to be added to make **car** a meaningful object in the manufacturing control context.

INFO

Data Objects and Object-Oriented Classes—Are They the Same Thing?

A common question occurs when data objects are discussed: Is a data object the same thing as an object-oriented[11] class? The answer is "no."

A data object defines a composite data item; that is, it incorporates a collection of individual data items (attributes) and gives the collection of items a name (the name of the data object).

An object-oriented class encapsulates data attributes but also incorporates the operations (methods) that manipulate the data implied by those attributes. In addition, the definition of classes implies a comprehensive infrastructure that is part of the object-oriented software engineering approach. Classes communicate with one another via messages, they can be organized into hierarchies, and they provide inheritance characteristics for objects that are an instance of a class.

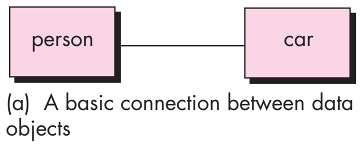
6.4.3 Relationships

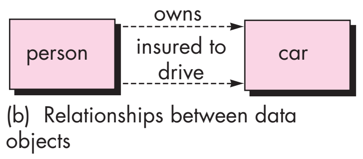
Data objects are connected to one another in different ways. Consider the two data objects, **person** and **car.** These objects can be represented using the simple notation

11 Readers who are unfamiliar with object-oriented concepts and terminology should refer to the brief tutorial presented in Appendix 2.

FIGURE 6.8

Relationships between data objects





KEY POINT

Relationships indicate the manner in which data objects are connected to one another.

illustrated in Figure 6.8a. A connection is established between **person** and **car** because the two objects are related. But what are the relationships? To determine the answer, you should understand the role of people (owners, in this case) and cars within the context of the software to be built. You can establish a set of object/relationship pairs that define the relevant relationships. For example,

- A person *owns* a car.
- A person *is insured to drive* a car.

The relationships *owns* and *insured to drive* define the relevant connections between **person** and **car.** Figure 6.8b illustrates these object-relationship pairs graphically. The arrows noted in Figure 6.8b provide important information about the directionality of the relationship and often reduce ambiguity or misinterpretations.

INFO

Entity-Relationship Diagrams

The object-relationship pair is the cornerstone of the data model. These pairs can be represented graphically using the entity-relationship diagram (ERD).[12] The ERD was originally proposed by Peter Chen [Che77] for the design of relational database systems and has been extended by others. A set of primary components is identified for the ERD: data objects, attributes, relationships, and various type indicators. The primary purpose of the ERD is to represent data objects and their relationships.

Rudimentary ERD notation has already been introduced. Data objects are represented by a labeled rectangle. Relationships are indicated with a labeled line connecting objects. In some variations of the ERD, the connecting line contains a diamond that is labeled with the relationship. Connections between data objects and relationships are established using a variety of special symbols that indicate cardinality and modality.[13] If you desire further information about data modeling and the entity-relationship diagram, see [Hob06] or [Sim05].

12 Although the ERD is still used in some database design applications, UML notation (Appendix 1) can now be used for data design.

13 The *cardinality* of an object-relationship pair specifies "the number of occurrences of one [object] that can be related to the number of occurrences of another [object]" {Til93]. The *modality* of a relationship is 0 if there is no explicit need for the relationship to occur or the relationship is optional. The modality is 1 if an occurrence of the relationship is mandatory.

SOFTWARE TOOLS

Data Modeling

Objective: Data modeling tools provide a software engineer with the ability to represent data objects, their characteristics, and their relationships. Used primarily for large database applications and other information systems projects, data modeling tools provide an automated means for creating comprehensive entity-relation diagrams, data object dictionaries, and related models.

Mechanics: Tools in this category enable the user to describe data objects and their relationships. In some cases, the tools use ERD notation. In others, the tools model relations using some other mechanism. Tools in this category are often used as part of database design and enable the creation of a database model by generating a database schema for common database management systems (DBMS).

Representative Tools:[14]

AllFusion ERWin, developed by Computer Associates (**www3.ca.com**), assists in the design of data objects, proper structure, and key elements for databases.

ER/Studio, developed by Embarcadero Software (**www.embarcadero.com**), supports entity-relationship modeling.

Oracle Designer, developed by Oracle Systems (**www.oracle.com**), "models business processes, data entities and relationships [that] are transformed into designs from which complete applications and databases are generated."

Visible Analyst, developed by Visible Systems (**www.visible.com**), supports a variety of analysis modeling functions including data modeling.

6.5 Class-Based Modeling

Class-based modeling represents the objects that the system will manipulate, the operations (also called methods or services) that will be applied to the objects to effect the manipulation, relationships (some hierarchical) between the objects, and the collaborations that occur between the classes that are defined. The elements of a class-based model include classes and objects, attributes, operations, class-responsibility-collaborator (CRC) models, collaboration diagrams, and packages. The sections that follow present a series of informal guidelines that will assist in their identification and representation.

6.5.1 Identifying Analysis Classes

Quote:

"The really hard problem is discovering what are the right objects [classes] in the first place."

Carl Argila

If you look around a room, there is a set of physical objects that can be easily identified, classified, and defined (in terms of attributes and operations). But when you "look around" the problem space of a software application, the classes (and objects) may be more difficult to comprehend.

We can begin to identify classes by examining the usage scenarios developed as part of the requirements model and performing a "grammatical parse" [Abb83] on the use cases developed for the system to be built. Classes are determined by underlining each noun or noun phrase and entering it into a simple table. Synonyms should be noted. If the class (noun) is required to implement a solution, then it is part of the solution space; otherwise, if a class is necessary only to describe a solution, it is part of the problem space.

14 Tools noted here do not represent an endorsement, but rather a sampling of tools in this category. In most cases, tool names are trademarked by their respective developers.

But what should we look for once all of the nouns have been isolated? *Analysis classes* manifest themselves in one of the following ways:

? How do analysis classes manifest themselves as elements of the solution space?

- *External entities* (e.g., other systems, devices, people) that produce or consume information to be used by a computer-based system.
- *Things* (e.g., reports, displays, letters, signals) that are part of the information domain for the problem.
- *Occurrences or events* (e.g., a property transfer or the completion of a series of robot movements) that occur within the context of system operation.
- *Roles* (e.g., manager, engineer, salesperson) played by people who interact with the system.
- *Organizational units* (e.g., division, group, team) that are relevant to an application.
- *Places* (e.g., manufacturing floor or loading dock) that establish the context of the problem and the overall function of the system.
- *Structures* (e.g., sensors, four-wheeled vehicles, or computers) that define a class of objects or related classes of objects.

This categorization is but one of many that have been proposed in the literature.[15] For example, Budd [Bud96] suggests a taxonomy of classes that includes *producers* (sources) and *consumers* (sinks) of data, *data managers, view* or *observer classes*, and *helper classes*.

It is also important to note what classes or objects are not. In general, a class should never have an "imperative procedural name" [Cas89]. For example, if the developers of software for a medical imaging system defined an object with the name **InvertImage** or even **ImageInversion,** they would be making a subtle mistake. The **Image** obtained from the software could, of course, be a class (it is a thing that is part of the information domain). Inversion of the image is an operation that is applied to the object. It is likely that inversion would be defined as an operation for the object **Image,** but it would not be defined as a separate class to connote "image inversion." As Cashman [Cas89] states: "the intent of object-orientation is to encapsulate, but still keep separate, data and operations on the data."

To illustrate how analysis classes might be defined during the early stages of modeling, consider a grammatical parse (nouns are underlined, verbs italicized) for a processing narrative[16] for the *SafeHome* security function.

15 Another important categorization, defining entity, boundary, and controller classes, is discussed in Section 6.5.4.

16 A processing narrative is similar to the use case in style but somewhat different in purpose. The processing narrative provides an overall description of the function to be developed. It is not a scenario written from one actor's point of view. It is important to note, however, that a grammatical parse can also be used for every use case developed as part of requirements gathering (elicitation).

The SafeHome security function *enables* the homeowner to *configure* the security system when it is *installed, monitors* all sensors *connected* to the security system, and *interacts* with the homeowner through the Internet, a PC, or a control panel.

During installation, the SafeHome PC is used to *program* and *configure* the system. Each sensor is assigned a number and type, a master password is programmed for *arming* and *disarming* the system, and telephone number(s) are *input* for *dialing* when a sensor event occurs.

ADVICE

The grammatical parse is not foolproof, but it can provide you with an excellent jump start, if you're struggling to define data objects and the transforms that operate on them.

When a sensor event is *recognized*, the software *invokes* an audible alarm attached to the system. After a delay time that is *specified* by the homeowner during system configuration activities, the software dials a telephone number of a monitoring service, *provides* information about the location, *reporting* the nature of the event that has been detected. The telephone number will be *redialed* every 20 seconds until telephone connection is *obtained.*

The homeowner *receives* security information via a control panel, the PC, or a browser, collectively called an interface. The interface *displays* prompting messages and system status information on the control panel, the PC ,or the browser window. Homeowner interaction takes the following form . . .

Extracting the nouns, we can propose a number of potential classes:

Potential Class	**General Classification**
homeowner	role or external entity
sensor	external entity
control panel	external entity
installation	occurrence
system (alias security system)	thing
number, type	not objects, attributes of sensor
master password	thing
telephone number	thing
sensor event	occurrence
audible alarm	external entity
monitoring service	organizational unit or external entity

The list would be continued until all nouns in the processing narrative have been considered. Note that I call each entry in the list a potential object. You must consider each further before a final decision is made.

? How do I determine whether a potential class should, in fact, become an analysis class?

Coad and Yourdon [Coa91] suggest six selection characteristics that should be used as you consider each potential class for inclusion in the analysis model:

1. *Retained information.* The potential class will be useful during analysis only if information about it must be remembered so that the system can function.
2. *Needed services.* The potential class must have a set of identifiable operations that can change the value of its attributes in some way.

3. *Multiple attributes.* During requirement analysis, the focus should be on "major" information; a class with a single attribute may, in fact, be useful during design, but is probably better represented as an attribute of another class during the analysis activity.
4. *Common attributes.* A set of attributes can be defined for the potential class and these attributes apply to all instances of the class.
5. *Common operations.* A set of operations can be defined for the potential class and these operations apply to all instances of the class.
6. *Essential requirements.* External entities that appear in the problem space and produce or consume information essential to the operation of any solution for the system will almost always be defined as classes in the requirements model.

Quote:

"Classes struggle, some classes triumph, others are eliminated."

Mao Zedong

To be considered a legitimate class for inclusion in the requirements model, a potential object should satisfy all (or almost all) of these characteristics. The decision for inclusion of potential classes in the analysis model is somewhat subjective, and later evaluation may cause an object to be discarded or reinstated. However, the first step of class-based modeling is the definition of classes, and decisions (even subjective ones) must be made. With this in mind, you should apply the selection characteristics to the list of potential *SafeHome* classes:

Potential Class	Characteristic Number That Applies
homeowner	rejected: 1, 2 fail even though 6 applies
sensor	accepted: all apply
control panel	accepted: all apply
installation	rejected
system (alias security function)	accepted: all apply
number, type	rejected: 3 fails, attributes of sensor
master password	rejected: 3 fails
telephone number	rejected: 3 fails
sensor event	accepted: all apply
audible alarm	accepted: 2, 3, 4, 5, 6 apply
monitoring service	rejected: 1, 2 fail even though 6 applies

It should be noted that (1) the preceding list is not all-inclusive, additional classes would have to be added to complete the model; (2) some of the rejected potential classes will become attributes for those classes that were accepted (e.g., **number** and **type** are attributes of **Sensor,** and **master password** and **telephone number** may become attributes of **System**); (3) different statements of the problem might cause different "accept or reject" decisions to be made (e.g., if each homeowner had an individual password or was identified by voice print, the **Homeowner** class would satisfy characteristics 1 and 2 and would have been accepted).

6.5.2 Specifying Attributes

KEY POINT

Attributes are the set of data objects that fully define the class within the context of the problem.

Attributes describe a class that has been selected for inclusion in the requirements model. In essence, it is the attributes that define the class—that clarify what is meant by the class in the context of the problem space. For example, if we were to build a system that tracks baseball statistics for professional baseball players, the attributes of the class **Player** would be quite different than the attributes of the same class when it is used in the context of the professional baseball pension system. In the former, attributes such as **name, position, batting average, fielding percentage, years played,** and **games played** might be relevant. For the latter, some of these attributes would be meaningful, but others would be replaced (or augmented) by attributes like **average salary, credit toward full vesting, pension plan options chosen, mailing address,** and the like.

To develop a meaningful set of attributes for an analysis class, you should study each use case and select those "things" that reasonably "belong" to the class. In addition, the following question should be answered for each class: "What data items (composite and/or elementary) fully define this class in the context of the problem at hand?"

To illustrate, we consider the **System** class defined for *SafeHome.* A homeowner can configure the security function to reflect sensor information, alarm response information, activation/deactivation information, identification information, and so forth. We can represent these composite data items in the following manner:

identification information = system ID + verification phone number + system status

alarm response information = delay time + telephone number

activation/deactivation information = master password + number of allowable tries + temporary password

Each of the data items to the right of the equal sign could be further defined to an elementary level, but for our purposes, they constitute a reasonable list of attributes for the **System** class (shaded portion of Figure 6.9).

Sensors are part of the overall *SafeHome* system, and yet they are not listed as data items or as attributes in Figure 6.9. **Sensor** has already been defined as a class, and multiple **Sensor** objects will be associated with the **System** class. In general, we avoid defining an item as an attribute if more than one of the items is to be associated with the class.

ADVICE

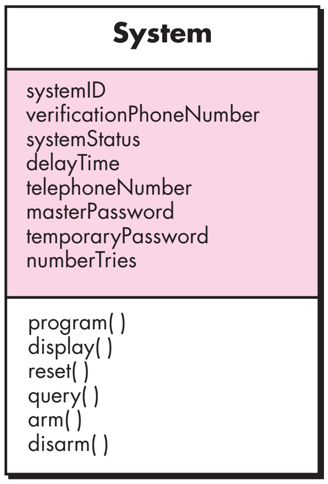
When you define operations for an analysis class, focus on problem-oriented behavior rather than behaviors required for implementation.

6.5.3 Defining Operations

Operations define the behavior of an object. Although many different types of operations exist, they can generally be divided into four broad categories: (1) operations that manipulate data in some way (e.g., adding, deleting, reformatting, selecting), (2) operations that perform a computation, (3) operations that inquire about the state

FIGURE 6.9

Class diagram for the system class



of an object, and (4) operations that monitor an object for the occurrence of a controlling event. These functions are accomplished by operating on attributes and/or associations (Section 6.5.5). Therefore, an operation must have "knowledge" of the nature of the class' attributes and associations.

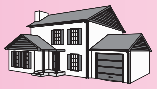
As a first iteration at deriving a set of operations for an analysis class, you can again study a processing narrative (or use case) and select those operations that reasonably belong to the class. To accomplish this, the grammatical parse is again studied and verbs are isolated. Some of these verbs will be legitimate operations and can be easily connected to a specific class. For example, from the *SafeHome* processing narrative presented earlier in this chapter, we see that "sensor is *assigned* a number and type" or "a master password is *programmed* for *arming and disarming* the system." These phrases indicate a number of things:

- That an *assign()* operation is relevant for the **Sensor** class.
- That a *program()* operation will be applied to the **System** class.
- That *arm()* and *disarm()* are operations that apply to **System** class.

Upon further investigation, it is likely that the operation *program()* will be divided into a number of more specific suboperations required to configure the system. For example, *program()* implies specifying phone numbers, configuring system characteristics (e.g., creating the sensor table, entering alarm characteristics), and entering password(s). But for now, we specify *program()* as a single operation.

In addition to the grammatical parse, you can gain additional insight into other operations by considering the communication that occurs between objects. Objects communicate by passing messages to one another. Before continuing with the specification of operations, I explore this matter in a bit more detail.

SafeHome



Class Models

The scene: Ed's cubicle, as requirements modeling begins.

The players: Jamie, Vinod, and Ed—all members of the *SafeHome* software engineering team.

The conversation:

[Ed has been working to extract classes from the use case template for ACS-DCV (presented in an earlier sidebar in this chapter) and is presenting the classes he has extracted to his colleagues.]

Ed: So when the homeowner wants to pick a camera, he or she has to pick it from a floor plan. I've defined a **FloorPlan** class. Here's the diagram.

(They look at Figure 6.10.)

Jamie: So **FloorPlan** is an object that is put together with walls, doors, windows, and cameras. That's what those labeled lines mean, right?

Ed: Yeah, they're called "associations." One class is associated with another according to the associations I've shown. [Associations are discussed in Section 6.5.5.]

Vinod: So the actual floor plan is made up of walls and contains cameras and sensors that are placed within those walls. How does the floor plan know where to put those objects?

Ed: It doesn't, but the other classes do. See the attributes under, say, **WallSegment,** which is used to build a wall. The wall segment has start and stop coordinates and the *draw()* operation does the rest.

Jamie: And the same goes for windows and doors. Looks like camera has a few extra attributes.

Ed: Yeah, I need them to provide pan and zoom info.

Vinod: I have a question. Why does the camera have an ID but the others don't? I notice you have an attribute called **nextWall.** How will **WallSegment** know what the next wall will be?

Ed: Good question, but as they say, that's a design decision, so I'm going to delay that until . . .

Jamie: Give me a break . . . I'll bet you've already figured it out.

Ed (smiling sheepishly): True, I'm gonna use a list structure which I'll model when we get to design. If you get religious about separating analysis and design, the level of detail I have right here could be suspect.

Jamie: Looks pretty good to me, but I have a few more questions.

(Jamie asks questions which result in minor modifications)

Vinod: Do you have CRC cards for each of the objects? If so, we ought to role-play through them, just to make sure nothing has been omitted.

Ed: I'm not quite sure how to do them.

Vinod: It's not hard and they really pay off. I'll show you.

Quote:

"One purpose of CRC cards is to fail early, to fail often, and to fail inexpensively. It is a lot cheaper to tear up a bunch of cards than it would be to reorganize a large amount of source code."

C. Horstmann

6.5.4 Class-Responsibility-Collaborator (CRC) Modeling

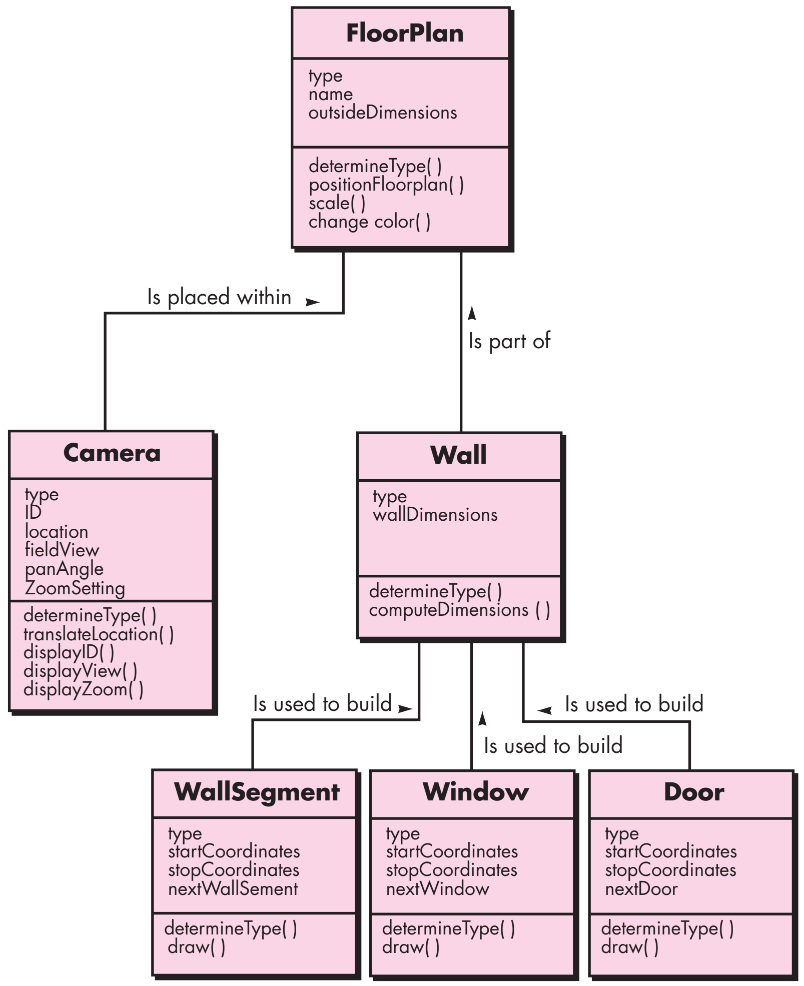
Class-responsibility-collaborator (CRC) modeling [Wir90] provides a simple means for identifying and organizing the classes that are relevant to system or product requirements. Ambler [Amb95] describes CRC modeling in the following way:

> A CRC model is really a collection of standard index cards that represent classes. The cards are divided into three sections. Along the top of the card you write the name of the class. In the body of the card you list the class responsibilities on the left and the collaborators on the right.

In reality, the CRC model may make use of actual or virtual index cards. The intent is to develop an organized representation of classes. *Responsibilities* are the attributes and operations that are relevant for the class. Stated simply, a responsibility is "anything the class knows or does" [Amb95]. *Collaborators* are those classes that are

FIGURE 6.10

Class diagram for FloorPlan (see sidebar discussion)



required to provide a class with the information needed to complete a responsibility. In general, a *collaboration* implies either a request for information or a request for some action.

A simple CRC index card for the **FloorPlan** class is illustrated in Figure 6.11. The list of responsibilities shown on the CRC card is preliminary and subject to additions or modification. The classes **Wall** and **Camera** are noted next to the responsibility that will require their collaboration.

WebRef

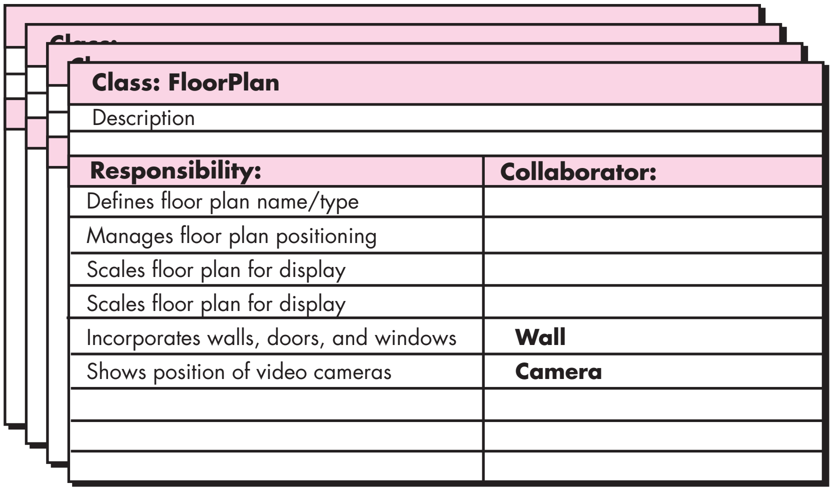
An excellent discussion of these class types can be found at **www.theumlcafe.com/a0079.htm**.

Classes. Basic guidelines for identifying classes and objects were presented earlier in this chapter. The taxonomy of class types presented in Section 6.5.1 can be extended by considering the following categories:

- *Entity classes*, also called *model* or *business* classes, are extracted directly from the statement of the problem (e.g., **FloorPlan** and **Sensor**). These

FIGURE 6.11

A CRC model index card



Class: FloorPlan

Description

Responsibility:	Collaborator:
Defines floor plan name/type	
Manages floor plan positioning	
Scales floor plan for display	
Scales floor plan for display	
Incorporates walls, doors, and windows	**Wall**
Shows position of video cameras	**Camera**

uote:

"Objects can be classified scientifically into three major categories: those that don't work, those that break down, and those that get lost."

Russell Baker

classes typically represent things that are to be stored in a database and persist throughout the duration of the application (unless they are specifically deleted).

- *Boundary classes* are used to create the interface (e.g., interactive screen or printed reports) that the user sees and interacts with as the software is used. Entity objects contain information that is important to users, but they do not display themselves. Boundary classes are designed with the responsibility of managing the way entity objects are represented to users. For example, a boundary class called **CameraWindow** would have the responsibility of displaying surveillance camera output for the *SafeHome* system.
- *Controller classes* manage a "unit of work" [UML03] from start to finish. That is, controller classes can be designed to manage (1) the creation or update of entity objects, (2) the instantiation of boundary objects as they obtain information from entity objects, (3) complex communication between sets of objects, (4) validation of data communicated between objects or between the user and the application. In general, controller classes are not considered until the design activity has begun.

? What guidelines can be applied for allocating responsibilities to classes?

Responsibilities. Basic guidelines for identifying responsibilities (attributes and operations) have been presented in Sections 6.5.2 and 6.5.3. Wirfs-Brock and her colleagues [Wir90] suggest five guidelines for allocating responsibilities to classes:

1. **System intelligence should be distributed across classes to best address the needs of the problem.** Every application encompasses a certain degree of intelligence; that is, what the system knows and what it can do. This intelligence can be distributed across classes in a number of

different ways. “Dumb” classes (those that have few responsibilities) can be modeled to act as servants to a few “smart” classes (those having many responsibilities). Although this approach makes the flow of control in a system straightforward, it has a few disadvantages: it concentrates all intelligence within a few classes, making changes more difficult, and it tends to require more classes, hence more development effort.

If system intelligence is more evenly distributed across the classes in an application, each object knows about and does only a few things (that are generally well focused), the cohesiveness of the system is improved.[17] This enhances the maintainability of the software and reduces the impact of side effects due to change.

To determine whether system intelligence is properly distributed, the responsibilities noted on each CRC model index card should be evaluated to determine if any class has an extraordinarily long list of responsibilities. This indicates a concentration of intelligence.[18] In addition, the responsibilities for each class should exhibit the same level of abstraction. For example, among the operations listed for an aggregate class called **CheckingAccount** a reviewer notes two responsibilities: *balance-the-account* and *check-off-cleared-checks.* The first operation (responsibility) implies a complex mathematical and logical procedure. The second is a simple clerical activity. Since these two operations are not at the same level of abstraction, *check-off-cleared-checks* should be placed within the responsibilities of **CheckEntry,** a class that is encompassed by the aggregate class **CheckingAccount.**

2. **Each responsibility should be stated as generally as possible.** This guideline implies that general responsibilities (both attributes and operations) should reside high in the class hierarchy (because they are generic, they will apply to all subclasses).
3. **Information and the behavior related to it should reside within the same class.** This achieves the object-oriented principle called *encapsulation*. Data and the processes that manipulate the data should be packaged as a cohesive unit.
4. **Information about one thing should be localized with a single class, not distributed across multiple classes.** A single class should take on the responsibility for storing and manipulating a specific type of information. This responsibility should not, in general, be shared across a number of classes. If information is distributed, software becomes more difficult to maintain and more challenging to test.

17 Cohesiveness is a design concept that is discussed in Chapter 8.
18 In such cases, it may be necessary to spit the class into multiple classes or complete subsystems in order to distribute intelligence more effectively.

5. **Responsibilities should be shared among related classes, when appropriate.** There are many cases in which a variety of related objects must all exhibit the same behavior at the same time. As an example, consider a video game that must display the following classes: **Player, PlayerBody, PlayerArms, PlayerLegs, PlayerHead.** Each of these classes has its own attributes (e.g., **position, orientation, color, speed**) and all must be updated and displayed as the user manipulates a joystick. The responsibilities *update()* and *display()* must therefore be shared by each of the objects noted. **Player** knows when something has changed and *update()* is required. It collaborates with the other objects to achieve a new position or orientation, but each object controls its own display.

Collaborations. Classes fulfill their responsibilities in one of two ways: (1) A class can use its own operations to manipulate its own attributes, thereby fulfilling a particular responsibility, or (2) a class can collaborate with other classes. Wirfs-Brock and her colleagues [Wir90] define collaborations in the following way:

> Collaborations represent requests from a client to a server in fulfillment of a client responsibility. A collaboration is the embodiment of the contract between the client and the server. . . . We say that an object collaborates with another object if, to fulfill a responsibility, it needs to send the other object any messages. A single collaboration flows in one direction—representing a request from the client to the server. From the client's point of view, each of its collaborations is associated with a particular responsibility implemented by the server.

Collaborations are identified by determining whether a class can fulfill each responsibility itself. If it cannot, then it needs to interact with another class. Hence, a collaboration.

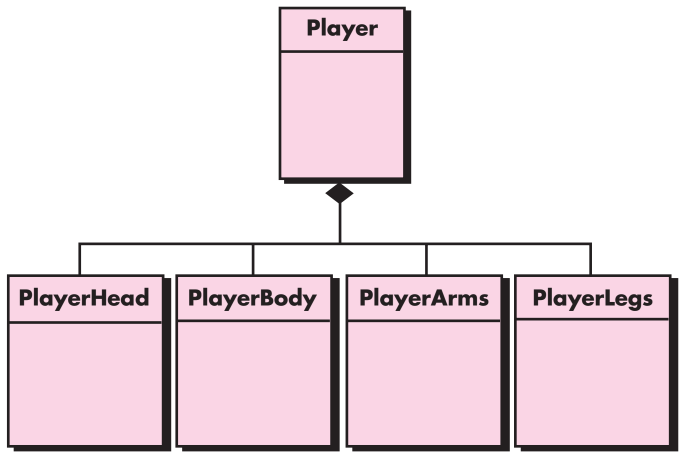
As an example, consider the *SafeHome* security function. As part of the activation procedure, the **ControlPanel** object must determine whether any sensors are open. A responsibility named *determine-sensor-status()* is defined. If sensors are open, **ControlPanel** must set a **status** attribute to "not ready." Sensor information can be acquired from each **Sensor** object. Therefore, the responsibility *determine-sensor-status()* can be fulfilled only if **ControlPanel** works in collaboration with **Sensor.**

To help in the identification of collaborators, you can examine three different generic relationships between classes [Wir90]: (1) the *is-part-of* relationship, (2) the *has-knowledge-of* relationship, and (3) the *depends-upon* relationship. Each of the three generic relationships is considered briefly in the paragraphs that follow.

All classes that are part of an aggregate class are connected to the aggregate class via an *is-part-of* relationship. Consider the classes defined for the video game noted earlier, the class **PlayerBody** *is-part-of* **Player,** as are **PlayerArms, PlayerLegs,** and **PlayerHead.** In UML, these relationships are represented as the aggregation shown in Figure 6.12.

FIGURE 6.12

A composite aggregate class



When one class must acquire information from another class, the *has-knowledge-of* relationship is established. The *determine-sensor-status()* responsibility noted earlier is an example of a *has-knowledge-of* relationship.

The *depends-upon* relationship implies that two classes have a dependency that is not achieved by *has-knowledge-of* or *is-part-of.* For example, **PlayerHead** must always be connected to **PlayerBody** (unless the video game is particularly violent), yet each object could exist without direct knowledge of the other. An attribute of the **PlayerHead** object called **center-position** is determined from the center position of **PlayerBody.** This information is obtained via a third object, **Player,** that acquires it from **PlayerBody.** Hence, **PlayerHead** *depends-upon* **PlayerBody.**

In all cases, the collaborator class name is recorded on the CRC model index card next to the responsibility that has spawned the collaboration. Therefore, the index card contains a list of responsibilities and the corresponding collaborations that enable the responsibilities to be fulfilled (Figure 6.11).

When a complete CRC model has been developed, stakeholders can review the model using the following approach [Amb95]:

1. All participants in the review (of the CRC model) are given a subset of the CRC model index cards. Cards that collaborate should be separated (i.e., no reviewer should have two cards that collaborate).
2. All use-case scenarios (and corresponding use-case diagrams) should be organized into categories.
3. The review leader reads the use case deliberately. As the review leader comes to a named object, she passes a token to the person holding the corresponding class index card. For example, a use case for *SafeHome* contains the following narrative:

   The homeowner observes the *SafeHome* control panel to determine if the system is ready for input. If the system is not ready, the homeowner must physically close windows/doors so that the ready indicator is present. [A not-ready indicator implies that a sensor is open, i.e., that a door or window is open.]

When the review leader comes to "control panel," in the use case narrative, the token is passed to the person holding the **ControlPanel** index card. The phrase "implies that a sensor is open" requires that the index card contains a responsibility that will validate this implication (the responsibility *determine-sensor-status()* accomplishes this). Next to the responsibility on the index card is the collaborator **Sensor.** The token is then passed to the **Sensor** object.

4. When the token is passed, the holder of the **Sensor** card is asked to describe the responsibilities noted on the card. The group determines whether one (or more) of the responsibilities satisfies the use-case requirement.
5. If the responsibilities and collaborations noted on the index cards cannot accommodate the use case, modifications are made to the cards. This may include the definition of new classes (and corresponding CRC index cards) or the specification of new or revised responsibilities or collaborations on existing cards.

This modus operandi continues until the use case is finished. When all use cases have been reviewed, requirements modeling continues.

SafeHome

CRC Models

The scene: Ed's cubicle, as requirements modeling begins.

The players: Vinod and Ed—members of the *SafeHome* software engineering team.

The conversation:

[Vinod has decided to show Ed how to develop CRC cards by showing him an example.]

Vinod: While you've been working on surveillance and Jamie has been tied up with security, I've been working on the home management function.

Ed: What's the status of that? Marketing kept changing its mind.

Vinod: Here's the first-cut use case for the whole function . . . we've refined it a bit, but it should give you an overall view . . .

Use case: *SafeHome* home management function.

Narrative: We want to use the home management interface on a PC or an Internet connection to control electronic devices that have wireless interface controllers. The system should allow me to turn specific lights on and off, to control appliances that are connected to a wireless interface, to set my heating and air conditioning system to temperatures that I define. To do this, I want to select the devices from a floor plan of the house. Each device must be identified on the floor plan. As an optional feature, I want to control all audiovisual devices—audio, television, DVD, digital recorders, and so forth.

With a single selection, I want to be able to set the entire house for various situations. One is *home,* another is *away,* a third is *overnight travel,* and a fourth is *extended travel.* All of these situations will have settings that will be applied to all devices. In the *overnight travel* and *extended travel* states, the system should turn lights on and off at random intervals (to make it look like someone is home) and control the heating and air conditioning system. I should be able to override these setting via the Internet with appropriate password protection . . .

Ed: The hardware guys have got all the wireless interfacing figured out?

Vinod (smiling): They're working on it; say it's no problem. Anyway, I extracted a bunch of classes for home management and we can use one as an example. Let's use the **HomeManagementInterface** class.

Ed: Okay . . . so the responsibilities are what . . . the attributes and operations for the class and the collaborations are the classes that the responsibilities point to.

Vinod: I thought you didn't understand CRC.

Ed: Maybe a little, but go ahead.

Vinod: So here's my class definition for **HomeManagementInterface.**

Attributes:

optionsPanel—contains info on buttons that enable user to select functionality.

situationPanel—contains info on buttons that enable user to select situation.

floorplan—same as surveillance object but this one displays devices.

deviceIcons—info on icons representing lights, appliances, HVAC, etc.

devicePanels—simulation of appliance or device control panel; allows control.

Operations:

displayControl(), selectControl(), displaySituation(), select situation(), accessFloorplan(), selectDeviceIcon(), displayDevicePanel(), accessDevicePanel(), . . .

Class: HomeManagementInterface

Responsibility	**Collaborator**
displayControl()	**OptionsPanel** (class)
selectControl()	**OptionsPanel** (class)
displaySituation()	**SituationPanel** (class)
selectSituation()	**SituationPanel** (class)
accessFloorplan()	**FloorPlan** (class) . . .

. . .

Ed: So when the operation *accessFloorplan()* is invoked, it collaborates with the **FloorPlan** object just like the one we developed for surveillance. Wait, I have a description of it here. (They look at Figure 6.10.)

Vinod: Exactly. And if we wanted to review the entire class model, we could start with this index card, then go to the collaborator's index card, and from there to one of the collaborator's collaborators, and so on.

Ed: Good way to find omissions or errors.

Vinod: Yep.

6.5.5 Associations and Dependencies

KEY POINT

An association defines a relationship between classes. Multiplicity defines how many of one class are related to how many of another class.

In many instances, two analysis classes are related to one another in some fashion, much like two data objects may be related to one another (Section 6.4.3). In UML these relationships are called *associations.* Referring back to Figure 6.10, the **FloorPlan** class is defined by identifying a set of associations between **FloorPlan** and two other classes, **Camera** and **Wall.** The class **Wall** is associated with three classes that allow a wall to be constructed, **WallSegment, Window,** and **Door.**

In some cases, an association may be further defined by indicating *multiplicity*. Referring to Figure 6.10, a **Wall** object is constructed from one or more **WallSegment** objects. In addition, the **Wall** object may contain 0 or more **Window** objects and 0 or more **Door** objects. These multiplicity constraints are illustrated in Figure 6.13, where "one or more" is represented using 1. .*, and "0 or more" by 0 . .*. In UML, the asterisk indicates an unlimited upper bound on the range.[19]

19 Other multiplicity relations—one to one, one to many, many to many, one to a specified range with lower and upper limits, and others—may be indicated as part of an association.

FIGURE 6.13
Multiplicity

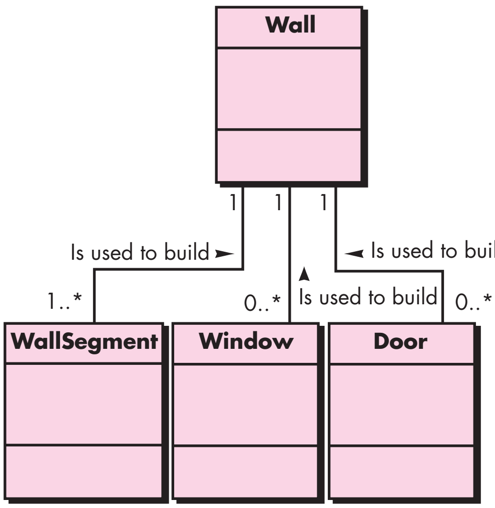
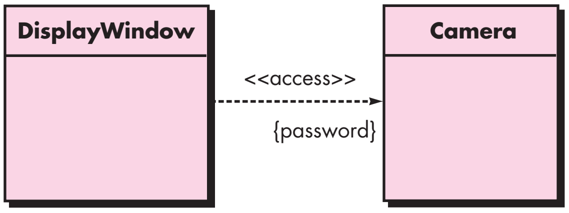


FIGURE 6.14
Dependencies



What is a stereotype?

In many instances, a client-server relationship exists between two analysis classes. In such cases, a client class depends on the server class in some way and a *dependency relationship* is established. Dependencies are defined by a stereotype. A *stereotype* is an "extensibility mechanism" [Arl02] within UML that allows you to define a special modeling element whose semantics are custom defined. In UML stereotypes are represented in double angle brackets (e.g., <<stereotype>>).

As an illustration of a simple dependency within the *SafeHome* surveillance system, a **Camera** object (in this case, the server class) provides a video image to a **DisplayWindow** object (in this case, the client class). The relationship between these two objects is not a simple association, yet a dependency association does exist. In a use case written for surveillance (not shown), you learn that a special password must be provided in order to view specific camera locations. One way to achieve this is to have **Camera** request a password and then grant permission to the **DisplayWindow** to produce the video display. This can be represented as shown in Figure 6.14 where <<access>> implies that the use of the camera output is controlled by a special password.

KEY POINT

A package is used to assemble a collection of related classes.

6.5.6 Analysis Packages

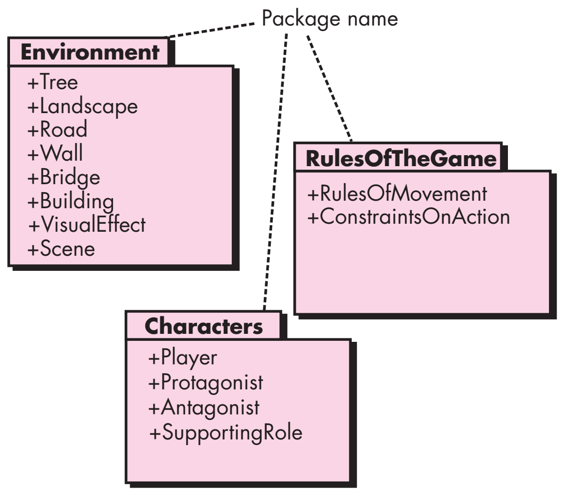
An important part of analysis modeling is categorization. That is, various elements of the analysis model (e.g., use cases, analysis classes) are categorized in a manner that packages them as a grouping—called an *analysis package*—that is given a representative name.

To illustrate the use of analysis packages, consider the video game that I introduced earlier. As the analysis model for the video game is developed, a large number of classes are derived. Some focus on the game environment—the visual scenes that the user sees as the game is played. Classes such as **Tree, Landscape, Road, Wall, Bridge, Building,** and **VisualEffect** might fall within this category. Others focus on the characters within the game, describing their physical features, actions, and constraints. Classes such as **Player** (described earlier), **Protagonist, Antagonist,** and **SupportingRoles** might be defined. Still others describe the rules of the game—how a player navigates through the environment. Classes such as **RulesOfMovement** and **ConstraintsOnAction** are candidates here. Many other categories might exist. These classes can be grouped in analysis packages as shown in Figure 6.15.

The plus sign preceding the analysis class name in each package indicates that the classes have public visibility and are therefore accessible from other packages. Although they are not shown in the figure, other symbols can precede an element within a package. A minus sign indicates that an element is hidden from all other packages and a # symbol indicates that an element is accessible only to packages contained within a given package.

FIGURE 6.15

Packages



6.6 Summary

The objective of requirements modeling is to create a variety of representations that describe what the customer requires, establish a basis for the creation of a software design, and define a set of requirements that can be validated once the software is built. The requirements model bridges the gap between a system-level representation that describes overall system and business functionality and a software design that describes the software's application architecture, user interface, and component-level structure.

Scenario-based models depict software requirements from the user's point of view. The use case—a narrative or template-driven description of an interaction between an actor and the software—is the primary modeling element. Derived during requirements elicitation, the use case defines the keys steps for a specific function or interaction. The degree of use-case formality and detail varies, but the end result provides necessary input to all other analysis modeling activities. Scenarios can also be described using an activity diagram—a flowchart-like graphical representation that depicts the processing flow within a specific scenario. Swimlane diagrams illustrate how the processing flow is allocated to various actors or classes.

Data modeling is used to describe the information space that will be constructed or manipulated by the software. Data modeling begins by representing data objects—composite information that must be understood by the software. The attributes of each data object are identified and relationships between data objects are described.

Class-based modeling uses information derived from scenario-based and data modeling elements to identify analysis classes. A grammatical parse may be used to extract candidate classes, attributes, and operations from text-based narratives. Criteria for the definition of a class are defined. A set of class-responsibility-collaborator index cards can be used to define relationships between classes. In addition, a variety of UML modeling notation can be applied to define hierarchies, relationships, associations, aggregations, and dependencies among classes. Analysis packages are used to categorize and group classes in a manner that makes them more manageable for large systems.

Problems and Points to Ponder

6.1. Is it possible to begin coding immediately after an analysis model has been created? Explain your answer and then argue the counterpoint.

6.2. An analysis rule of thumb is that the model "should focus on requirements that are visible within the problem or business domain." What types of requirements are *not* visible in these domains? Provide a few examples.

6.3. What is the purpose of domain analysis? How is it related to the concept of requirements patterns?