CHAPTER

8 REQUIREMENTS MODELING—A RECOMMENDED APPROACH

The written word is a wonderful vehicle for communication, but it is not necessarily the best way to represent the requirements for computer software. 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 is actually a set of models that make up the first technical representation of a system. Software engineers often prefer to include graphical representations of complex model relationships.

KEY CONCEPTS

QUICK LOOK

What is it? 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 these models from requirements elicited from various stakeholders.

Why is it important? Requirements models can be readily evaluated by all stakeholders, resulting in useful feedback at the earliest possible time. Later, as the model is refined, it becomes the basis for software design.

What are the steps? Requirements modeling combines three steps: scenario-based modeling, class modeling, and behavioral modeling.

What is the work product? Usage scenarios, called use cases, describe software functions and usage. In addition, a series of UML diagrams can be used to represent system behavior and other aspects.

How do I ensure that I've done it right? Requirements modeling work products must be reviewed for correctness, completeness, and consistency.

For some types of software, a user story (Section 7.3.2) may be the only requirements modeling representation that is required. For others, formal use cases (Section 7.4) and class-based models (Section 8.3) may be developed. 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. Class-based methods can be used to create a representation of an application that can be understood by nontechnical stakeholders.

In still other situations, complex application requirements may demand an examination of how an application behaves in reaction to either internal or external events. These behaviors need to be modeled (Section 8.5) as well. UML diagrams have become a standard software engineering means of modeling analysis model element relationships and behaviors graphically. As the requirements model is refined and expanding, it evolves into a specification that can be used by software engineers in the creation of the software design.

The important thing to keep in mind when modeling requirements is to only create the models that will be used by the development team. If models developed early in a requirements analysis phase of a project are not referred to during the design and implementation phases, they may not be worth updating. The sections that follow present a series of informal guidelines that will assist in the creation and representation of requirements models.

8.1 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 7).

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."
- *Class-oriented models* that represent object-oriented classes (attributes and operations) and how classes collaborate to achieve system requirements.
- *Behavioral models* that depict how the software reacts to internal or external "events."
- *Data models* that depict the information domain for the problem.
- *Flow-oriented models* that represent the functional elements of the system and how they transform data as they move through the system.

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 section, we focus on *scenario-based modeling*—a technique that is very popular throughout the software engineering community. In Sections 8.3 and 8.5 we consider class-based modeling and behavioral modeling. Over the past decade, flow and data modeling have become less commonly used, while scenario and class-based methods, supplemented with behavioral approaches have grown in popularity.[1]

8.1.1 Overall Objectives and Philosophy

Throughout analysis modeling, your primary focus is on *what,* not *how.* What user interaction occurs, 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 previous chapters, we 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 (software, hardware, data, human elements) and a software design (Chapters 9 through 14) that describes the software's application architecture, user interface, and component-level structure. This relationship is illustrated in Figure 8.1.

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 between analysis and design modeling tasks is not always possible. Some design invariably occurs as part of analysis, and some analysis will be conducted during design.

8.1.2 Analysis Rules of Thumb

Several rules of thumb [Arl02] are worth considering when creating an analysis model. First, focus on the problem or business domain while keeping the level of abstraction high. Second, recognize that an analysis model should provide insight into

---

1 A presentation of flow-oriented modeling and data modeling is no longer included in this chapter. However, copious information about these older requirements modeling methods can be found on the Web. If you have interest, use the search phrase "structured analysis."

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 4) to better understand requirements for the system.

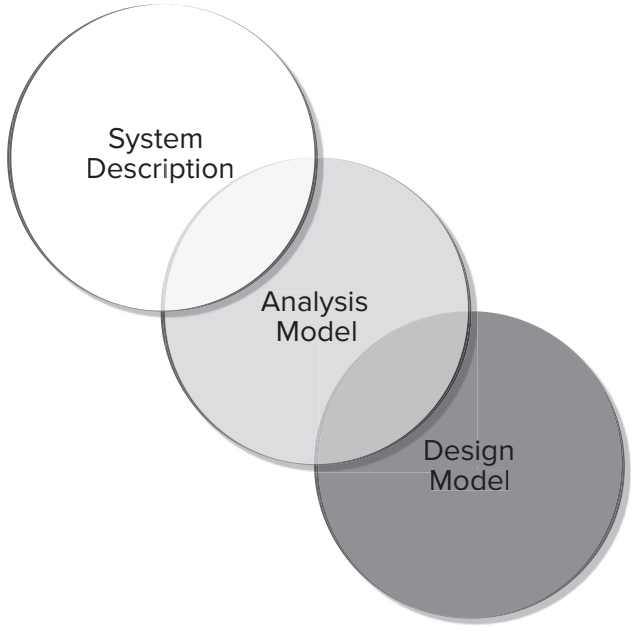


FIGURE 8.1
The analysis model as a bridge between the system description and the design model

the information domain, the function, and the behavior of the software. Third, delay a consideration of software architecture and nonfunctional details until later in the modeling activity. Also, it's important to be aware of the ways in which elements of the software are interconnnected with other elements (we call this *system coupling*).

The analysis model must be structured in a way that provides value to all stakeholders and should be kept as simple as possible without sacrificing clarity.

8.1.3 Requirements Modeling Principles

Over the past four decades, several requirements modeling methods have been developed. Investigators have identified requirements analysis problems and their causes and have developed a variety of modeling notations and corresponding sets of heuristics to overcome them. Each analysis method has a unique point of view. A set of operational principles relates analysis methods:

Principle 1. ***The information domain of a problem must be represented and understood.*** The *information domain* encompasses the data that flow into the system (from end users, other systems, or external devices), the data that flow out of the system (via the user interface, network interfaces, reports, graphics, and other means), and the data stores that collect and organize the data that are maintained permanently.

Principle 2. ***The functions that the software performs must be defined.*** Software functions provide direct benefit to end users and those that provide internal support for those features that are user visible. Some functions transform data that

flow into the system. In other cases, functions effect some level of control over internal software processing or external system elements.

Principle 3. ***The behavior of the software (as a consequence of external events) must be represented.*** The behavior of computer software is driven by its interaction with the external environment. Input provided by end users, control data provided by an external system, or monitoring data collected over a network all cause the software to behave in a specific way.

Principle 4. ***The models that depict information, function, and behavior must be partitioned in a manner that uncovers detail in a layered (or hierarchical) fashion.*** Requirements modeling is the first step in software engineering problem solving. It allows you to better understand the problem and establishes a basis for the solution (design). Complex problems are difficult to solve in their entirety. For this reason, you should use a divide-and-conquer strategy. A large, complex problem is divided into subproblems until each subproblem is relatively easy to understand. This concept is called *partitioning* or *separation of concerns,* and it is a key strategy in requirements modeling.

Principle 5. ***The analysis task should move from essential information toward implementation detail.*** Analysis modeling begins by describing the problem from the end-user's perspective. The "essence" of the problem is described without any consideration of how a solution will be implemented. For example, a video game requires that the player "instruct" its protagonist on what direction to proceed as she moves into a dangerous maze. That is the essence of the problem. Implementation detail (normally described as part of the design model) indicates how the essence will be implemented. For the video game, voice input might be used. Alternatively, a keyboard command might be typed, a game pad joystick (or mouse) might be pointed in a specific direction, a motion-sensitive device might be waved in the air, or a device that reads the player's body or eye movements directly can be used.

By applying these principles, a software engineer approaches a problem systematically. But how are these principles applied in practice? This question will be answered in the remainder of this chapter.

8.2 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. Using UML[4] to model requirements begins with the creation of scenarios in the form of use case diagrams, activity diagrams, and sequence diagrams.

4 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.

8.2.1 Actors and User Profiles

A UML *actor* models an entity that interacts with a system object. Actors may represent roles played by human stakeholders or external hardware as they interact with system objects by exchanging information. A single physical entity may be portrayed by several actors if the physical entity takes on several roles that are relevant to realizing different system functions.

A UML *profile* provides a way of extending an existing model to other domains or platforms. This might allow you to revise the model of a Web-based system and model the system for various mobile platforms. Profiles might also be used to model the system from the viewpoints of different users. For example, system administrators may have a different view of the functionality of an automated teller machine than end users.

8.2.2 Creating Use Cases

In Chapter 7, we discussed user stories as a way of summarizing the stakeholders' perspective on how they will interact with the proposed system. However, they are written in plain English or the language used by the stakeholders. Developers need more precise means of describing this interaction before beginning to create the software. Alistair Cockburn characterizes a *use case* as a "contract for behavior" [Coc01b]. As we discussed in Chapter 7, the "contract" defines the way in which an actor[5] uses a computer-based system to accomplish some goal. In other words, a use case captures the interactions that occur between producers and consumers of information within the system itself. In this section, we examine how preliminary use cases are developed as part of the analysis modeling activity.[6]

In Chapter 7, we 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 modeling tool.

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 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 8.4) developed as part of requirements modeling.

---

5 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].

6 Use cases are a particularly important part of analysis modeling for user interfaces. Interface analysis and design is discussed in detail in Chapter 12.

SAFEHOME

Developing Another Preliminary Use Case

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?

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 need to have the AR/VR 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 a mobile device 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 mobile device or 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 device 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 like the one proposed in Chapter 7.

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 user story:

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 mobile device 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," 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.

It is important to note that this sequential presentation does not consider any alternative interactions (the narrative is free flowing and did represent a few alternatives). Use cases of this type are sometimes referred to as *primary scenarios* [Sch98].

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 [Sch98]:

- *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."[7] 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 several 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.

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

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

an alternative chosen by the actor) that causes the system to exhibit somewhat different behavior.

Cockburn [Coc01b] recommends 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 the 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 or mobile 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 by asking and answering these questions should be "rationalized" [Coc01b] 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).

8.2.3 Documenting Use Cases

The informal use cases presented in Section 8.2.2 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 8.2.2). Additional headings may or may not be included and are reasonably self-explanatory.

Most developers like to create graphical representation as they create use cases out of user stories. A diagrammatic representation can facilitate better understanding of the problem by all stakeholders, particularly when the scenario is complex. As we noted earlier in this book, UML provides use case diagramming capability. Figure 8.2 depicts a use case diagram for the *SafeHome* product. The use case diagram helps to show the relations among the use cases in the usage scenario. Each use case is represented by an oval. Only the **ACS-DCV** use case has been discussed in this section.

Each modeling notation has limitations, and the UML 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

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 cameras 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" button 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: Infrequent.

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

Secondary actors: System administrator, cameras.

Channels to secondary actors:

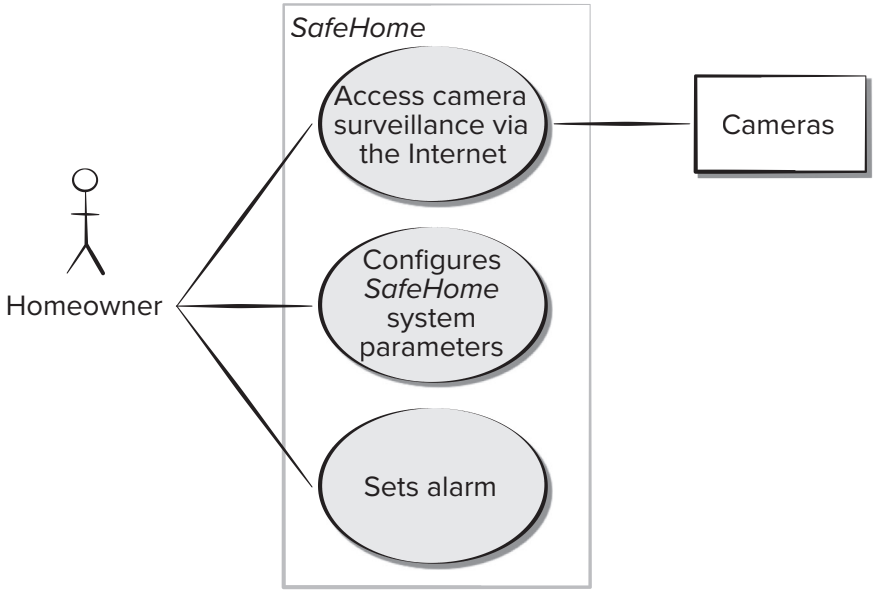
1. System administrator: PC-based system.
2. Cameras: wireless connectivity.

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?

FIGURE 8.2

Preliminary use case diagram for the *SafeHome* system



description is unclear, the use case can be misleading or ambiguous. A use case focuses on function 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.

8.3 CLASS-BASED MODELING

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.

8.3.1 Identifying Analysis Classes

We can begin to identify classes by examining the usage scenarios developed as part of the requirements model (Section 8.2) 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.

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

- *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.[8] 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*.

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[9] for the *SafeHome* security function.

> 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.
>
> 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 . . .

8 Another important categorization, defining entity, boundary, and controller classes, is discussed in Section 10.3.

9 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).

Extracting the nouns, we can propose several 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 we call each entry in the list a "potential" object. We must consider each further before a final decision is made.

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.

To be considered a legitimate class for inclusion in the requirements model, a potential object should satisfy most 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. 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, so 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**); and (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).

8.3.2 Defining Attributes and Operations

Attributes describe a class that has been selected for inclusion in the analysis model. It is the attributes that define the class—that clarify what is meant by the class in the context of the problem space.

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 and deactivation information, identification information, and so forth. We can represent these composite data items in the following manner:

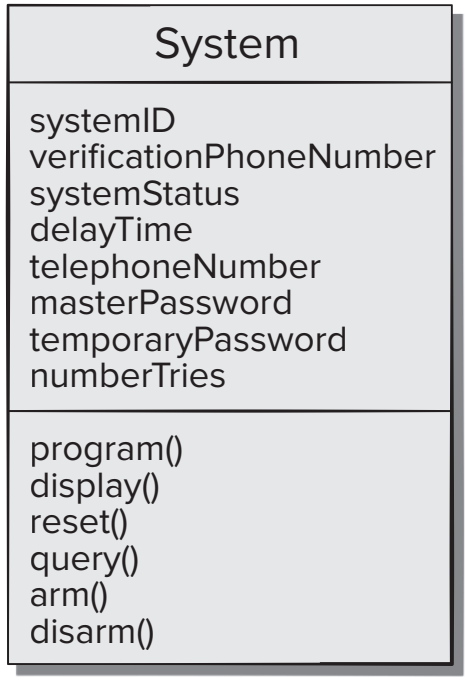
identification information = system ID + verification phone number + system status

alarm response information = delay time + telephone number

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

FIGURE 8.3

Class diagram for the System class



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 (Figure 8.3).

Sensors are part of the overall *SafeHome* system, and yet they are not listed as data items or as attributes in Figure 8.3. **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.

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 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 8.3.3). Therefore, an operation must have "knowledge" of the nature of the class attributes and associations.

8.3.3 UML Class Models

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 several 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.

SAFEHOME

Class Models

The scene: Ed's cubicle, as analysis 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 must pick it from a floor plan. I've defined a **FloorPlan** class. Here's the diagram.

(They look at Figure 8.4.)

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 8.3.3.]

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.)

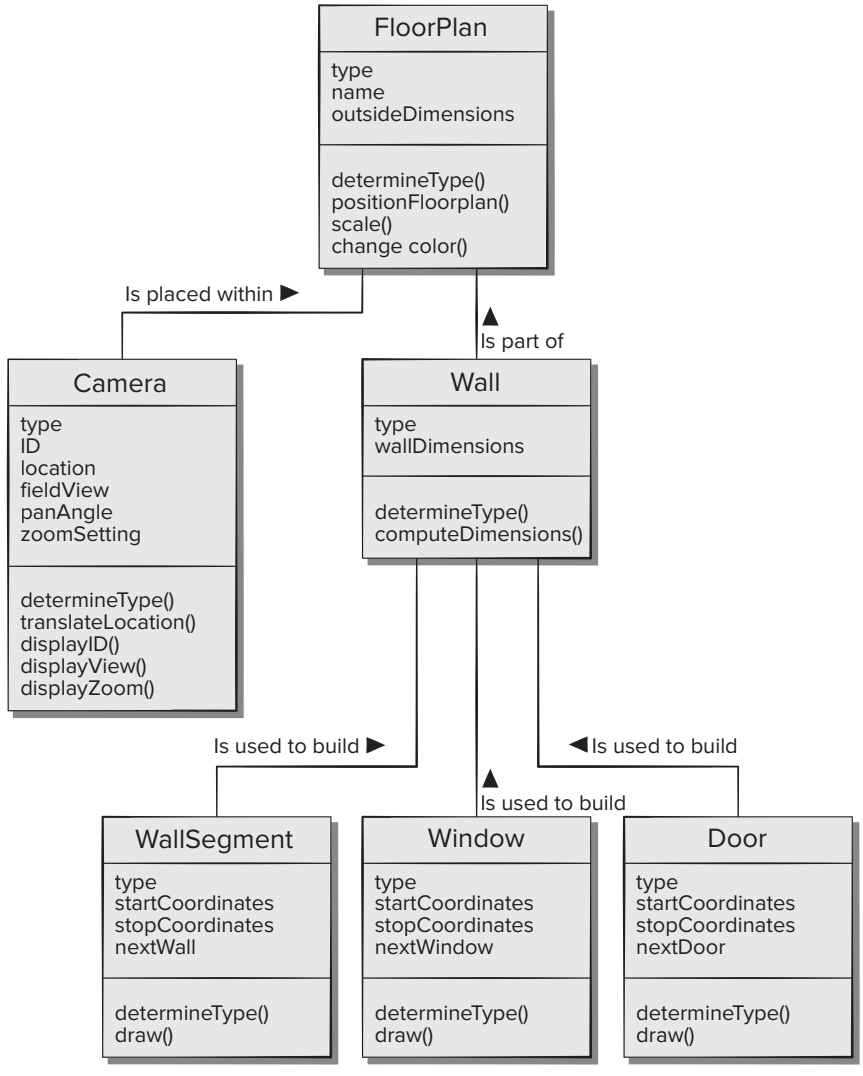
Vinod: Do you have CRC cards for each of the objects? If so, we ought to role-play through them, just to make sure nothing has been omitted.

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

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

FIGURE 8.4

Class diagram for FloorPlan (see sidebar discussion)



Upon further investigation, it is likely that the operation *program()* will be divided into several 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, we explore this matter in a bit more detail.

In many instances, two analysis classes are related to one another in some fashion. In UML, these relationships are called *associations*. Referring to Figure 8.4, 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.**

8.3.4 Class-Responsibility-Collaborator 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. A CRC model can be viewed as a collection of index cards. Each index card contains a list of responsibilities on the left side and the corresponding collaborations that enable the responsibilities to be fulfilled on the right side (Figure 8.5). *Responsibilities* are the attributes and operations that are relevant for the class. *Collaborators* are those classes that provide a class with the information needed or action required to complete a responsibility. A simple CRC index card for the **FloorPlan** class is illustrated in Figure 8.5.

The list of responsibilities shown on the CRC card is preliminary and is subject to additions or modification. The classes **Wall** and **Camera** are noted next to the responsibility that requires their collaboration.

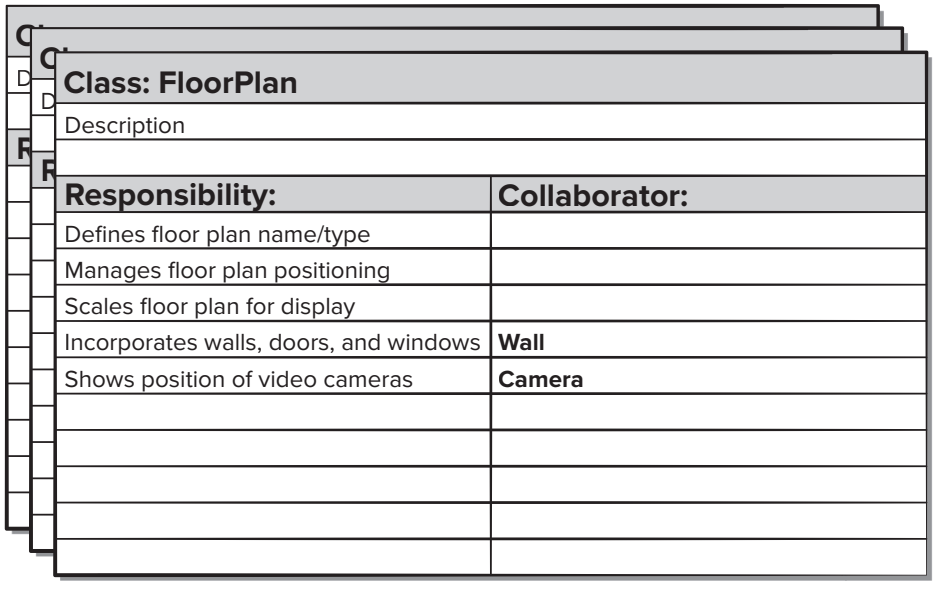
Classes. Basic guidelines for identifying classes and objects were presented earlier in Section 8.3.1.

Responsibilities. Basic guidelines for identifying responsibilities (attributes and operations) were presented in Section 8.3.2.

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

FIGURE 8.5

A CRC model index card



responsibility itself, or (2) a class can collaborate with other classes. Collaborations are identified by determining whether a class can fulfill each responsibility itself. If it cannot, then it needs to interact with another class.

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. The responsibility *determine-sensor-status()* can be fulfilled only if **ControlPanel** works in collaboration with **Sensor.**

Once a complete CRC model has been developed, the representatives from the 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. No reviewer should have two cards that collaborate.
2. 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.
3. When the token is passed, the holder of the class 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.
4. If an error is found, modifications are made to the cards. This may include the definition of new classes (and corresponding CRC index cards) or revising lists of responsibilities or collaborations on existing cards.

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 mobile device 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, and 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 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: Do the hardware guys have 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: So, looking at the **HomeManagementInterface** card, 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 8.4.)

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.

8.4 FUNCTIONAL MODELING

The *functional model* addresses two application processing elements, each representing a different level of procedural abstraction: (1) user-observable functionality that is delivered by the app to end users, and (2) the operations contained within analysis classes that implement behaviors associated with the class.

User-observable functionality encompasses any processing functions that are initiated directly by the user. For example, a financial mobile app might implement a variety of financial functions (e.g., computation of mortgage payment). These functions may be implemented using operations within analysis classes, but from the point of view of the end user, the function (more correctly, the data provided by the function) is the visible outcome.

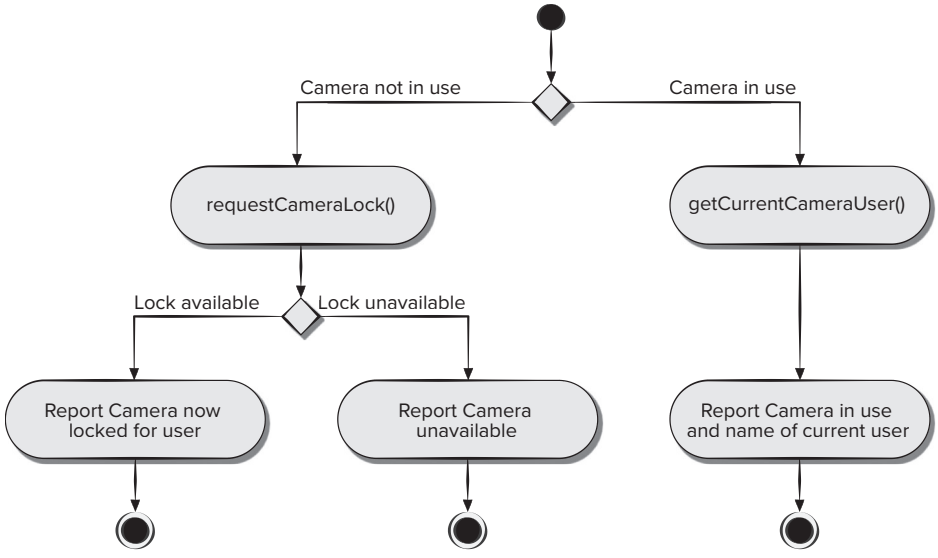
At a lower level of procedural abstraction, the requirements model describes the processing to be performed by analysis class operations. These operations manipulate class attributes and are involved as classes collaborate with one another to accomplish some required behavior.

8.4.1 A Procedural View

Regardless of the level of procedural abstraction, the UML activity diagram can be used to represent processing details. At the analysis level, activity diagrams should be used only where the functionality is relatively complex. Much of the complexity of mobile apps occurs not in the functionality provided, but rather with the nature of the information that can be accessed and the ways in which this can be manipulated.

FIGURE 8.6

Activity diagram for the *takeControlOfCamera()* operation



The UML activity diagram supplements the use case by providing a graphical representation of the flow of interaction within a specific scenario. An activity diagram is like a flowchart. The activity diagram (Figure 8.6) 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 example of a relatively complex functionality for **SafeHomeAssured.com** is addressed by a use case entitled *Get recommendations for sensor layout for my space.* The user has already developed a layout for the space to be monitored, and in this use case, selects that layout and requests recommended locations for sensors within the layout. **SafeHomeAssured.com** responds with a graphical representation of the layout with additional information on the recommended locations for sensors. The interaction is quite simple and the content is somewhat more complex, but the underlying functionality is very sophisticated. The system must undertake a relatively complex analysis of the floor layout to determine the optimal set of sensors. It must examine room dimensions and the location of doors and windows and coordinate these with sensor capabilities and specifications. No small task! A set of activity diagrams can be used to describe processing for this use case.

The second example is the use case *Control cameras.* In this use case, the interaction is relatively simple, but there is the potential for complex functionality, given that this "simple" operation requires complex communication with devices located remotely and accessible across the Internet. A further possible complication relates to negotiation of control when multiple authorized people attempt to monitor and/or control a single sensor at the same time.

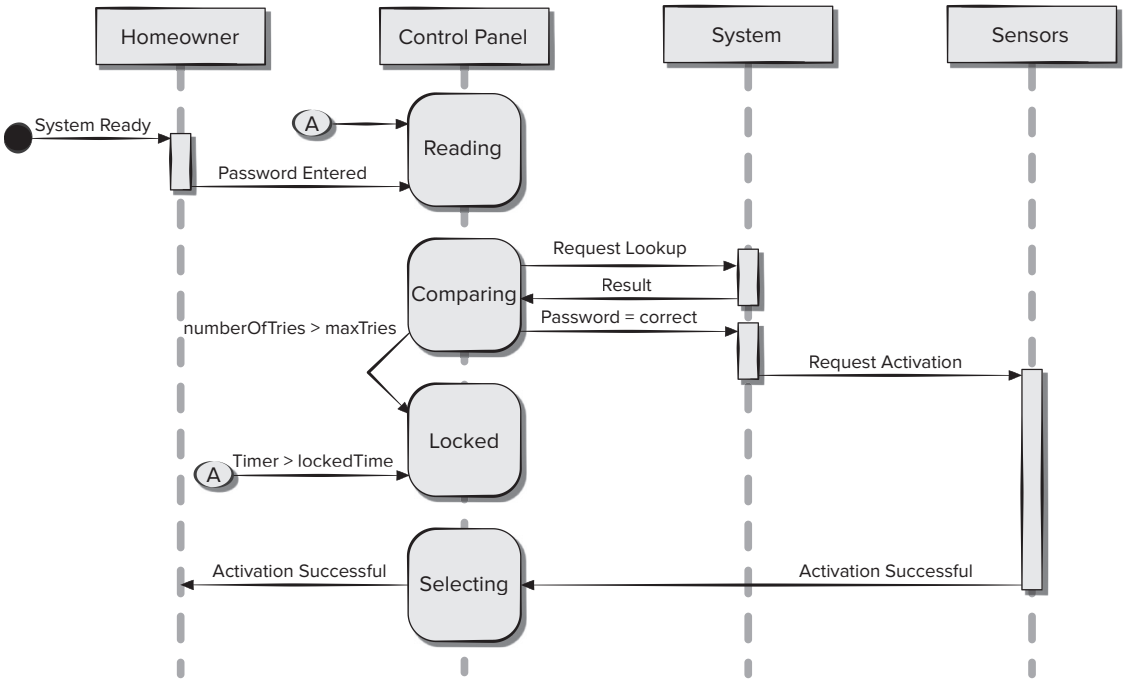
Figure 8.6 depicts an activity diagram for the *takeControlOfCamera()* operation that is part of the **Camera** analysis class used within the *Control cameras* use case. It should be noted that two additional operations are invoked with the procedural flow: *requestCameraLock(),* which tries to lock the camera for this user, and *getCurrentCameraUser(),* which retrieves the name of the user who is currently controlling the camera. The construction details indicating how these operations are invoked and the interface details for each operation are not considered until software design commences.

8.4.2 UML Sequence Diagrams

The UML *sequence diagram* can be used for behavioral modeling. Sequence diagrams can also be used to show how events cause transitions from object to object. Once events have been identified by examining a use case, the modeler creates a sequence diagram—a representation of how events cause flow from one object to another as a function of time. The sequence diagram is a shorthand version of the use case. It represents key classes and the events that cause behavior to flow from class to class.

Figure 8.7 illustrates a partial sequence diagram for the *SafeHome* security function. Each of the arrows represents an event (derived from a use case) and indicates how the event channels behavior between *SafeHome* objects. Time is measured vertically (downward), and the narrow vertical rectangles represent time spent in processing an activity. States may be shown along a vertical time line.

FIGURE 8.7 **Sequence diagram (partial) for the *SafeHome* security function**



The first event, *system ready*, is derived from the external environment and channels behavior to the **Homeowner** object. The homeowner enters a password. A *request lookup* event is passed to **System,** which looks up the password in a simple database and returns a *result* (*found* or *not found*) to **ControlPanel** (now in the *comparing* state). A valid password results in a *password=correct* event to **System,** which activates **Sensors** with a *request activation* event. Ultimately, control is passed back to the homeowner with the *activation successful* event.

Once a complete sequence diagram has been developed, all the events that cause transitions between system objects can be collated into a set of input events and output events (from an object). This information is useful in the creation of an effective design for the system to be built.

8.5 BEHAVIORAL MODELING

A *behavioral model* indicates how software will respond to internal or external events or stimuli. This information is useful in the creation of an effective design for the system to be built. UML activity diagrams can be used to model how system elements respond to internal events. UML state diagrams can be used to model how system elements respond to external events.

To create the model, you should perform the following steps: (1) evaluate all use cases to fully understand the sequence of interaction within the system, (2) identify events that drive the interaction sequence and understand how these events relate to specific objects, (3) create a sequence for each use case, (4) build a state diagram for the system, and (5) review the behavioral model to verify accuracy and consistency. Each of these steps is discussed in the sections that follow.

8.5.1 Identifying Events with the Use Case

In Section 8.3.3, you learned that the use case represents a sequence of activities that involves actors and the system. In general, an event occurs whenever the system and an actor exchange information. An event is *not* the information that has been exchanged, but rather the fact that information has been exchanged.

A use case is examined for points of information exchange. To illustrate, reconsider the use case for a portion of the *SafeHome* security function.

> The homeowner uses the keypad to key in a four-digit password. The password is compared with the valid password stored in the system. If the password is incorrect, the control panel will beep once and reset itself for additional input. If the password is correct, the control panel awaits further action.

The underlined portions of the use case scenario indicate events. An actor should be identified for each event; the information that is exchanged should be noted, and any conditions or constraints should be listed.

As an example of a typical event, consider the underlined use case phrase "homeowner uses the keypad to key in a four-digit password." In the context of the requirements model, the object, **Homeowner,**[10] transmits an event to the object **ControlPanel.**

10 In this example, we assume that each user (homeowner) that interacts with *SafeHome* has an identifying password and is therefore a legitimate object.

The event might be called *password entered*. The information transferred is the four digits that constitute the password, but this is not an essential part of the behavioral model. It is important to note that some events have an explicit impact on the flow of control of the use case, while others have no direct impact on the flow of control. For example, the event *password entered* does not explicitly change the flow of control of the use case, but the results of the event *password compared* (derived from the interaction "password is compared with the valid password stored in the system") will have an explicit impact on the information and control flow of the *SafeHome* software.

Once all events have been identified, they are allocated to the objects involved. Objects can be responsible for generating events (e.g., **Homeowner** generates the *password entered* event) or recognizing events that have occurred elsewhere (e.g., **ControlPanel** recognizes the binary result of the *password compared* event).

8.5.2 UML State Diagrams

In the context of behavioral modeling, two different characterizations of states must be considered: (1) the state of each class as the system performs its function and (2) the state of the system as observed from the outside as the system performs its function.

The state of a class takes on both passive and active characteristics [Cha93]. A *passive state* is simply the current values assigned to an object's attributes. The *active state* of an object indicates the status of the object as it undergoes a continuing transformation or processing. An *event* (sometimes called a *trigger*) must occur to force an object to make a transition from one active state to another.

State Diagrams for Analysis Classes. One component of a behavioral model is a UML state diagram[11] that represents active states for each class and the events (triggers) that cause changes between these active states. Figure 8.8 illustrates a state diagram for the **ControlPanel** object in the *SafeHome* security function.

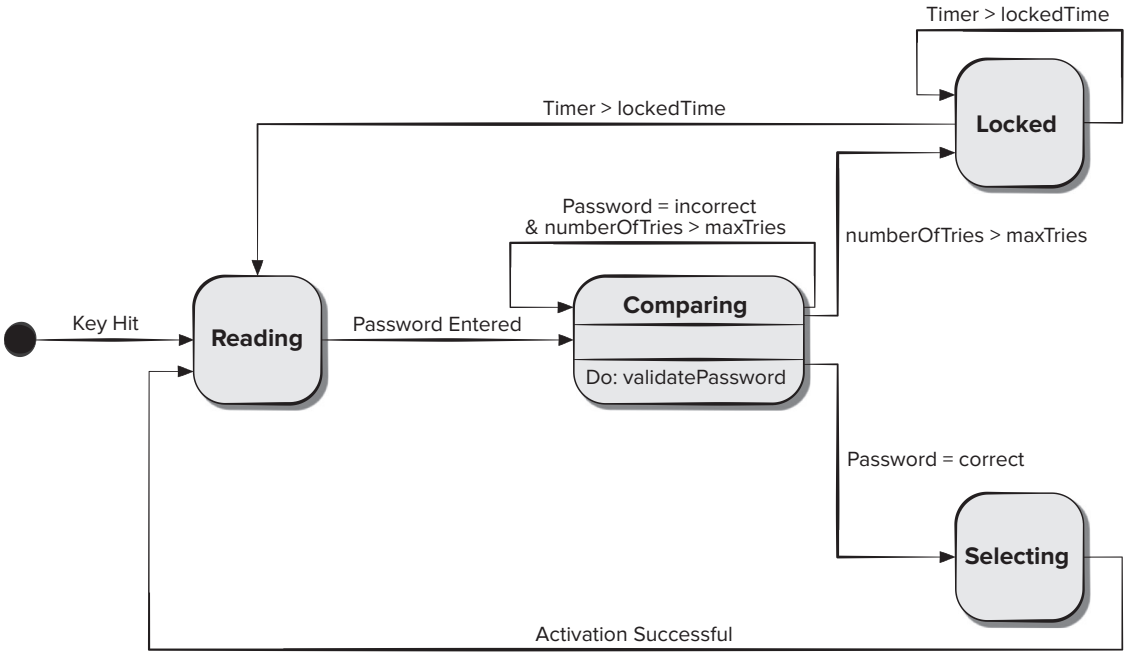
Each arrow shown in Figure 8.8 represents a transition from one active state of an object to another. The labels shown for each arrow represent the event that triggers the transition. Although the active state model provides useful insight into the "life history" of an object, it is possible to specify additional information to provide more depth in understanding the behavior of an object. In addition to specifying the event that causes the transition to occur, you can specify a guard and an action [Cha93]. A *guard* is a Boolean condition that must be satisfied for the transition to occur. For example, the guard for the transition from the "reading" state to the "comparing" state in Figure 8.8 can be determined by examining the use case:

```
if (password input = 4 digits) then compare to stored password
```

In general, the guard for a transition usually depends upon the value of one or more attributes of an object. In other words, the guard depends on the passive state of the object.

11 If you are unfamiliar with UML, a brief introduction to this important modeling notation is presented in Appendix 1.

FIGURE 8.8 State diagram for the ControlPanel class



An *action* occurs concurrently with the state transition or because of it and generally involves one or more operations (responsibilities) of the object. For example, the action connected to the *password entered* event (Figure 8.8) is an operation named *validatePassword()* that accesses a **password** object and performs a digit-by-digit comparison to validate the entered password.

8.5.3 UML Activity Diagrams

The UML activity diagram supplements the use case by providing a graphical representation of the flow of interaction within a specific scenario. Many software engineers like to describe activity diagrams as a way of representing how a system reacts to internal events.

An activity diagram for the **ACS-DCV** use case is shown in Figure 8.9. It should be noted that the activity diagram adds additional detail not directly mentioned (but implied) by the use case. For example, a user may only attempt to enter **userID** and **password** a limited number of times. A decision diamond represents this below: "Prompt for reentry."

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 (Section 8.3.1) 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.

FIGURE 8.9

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

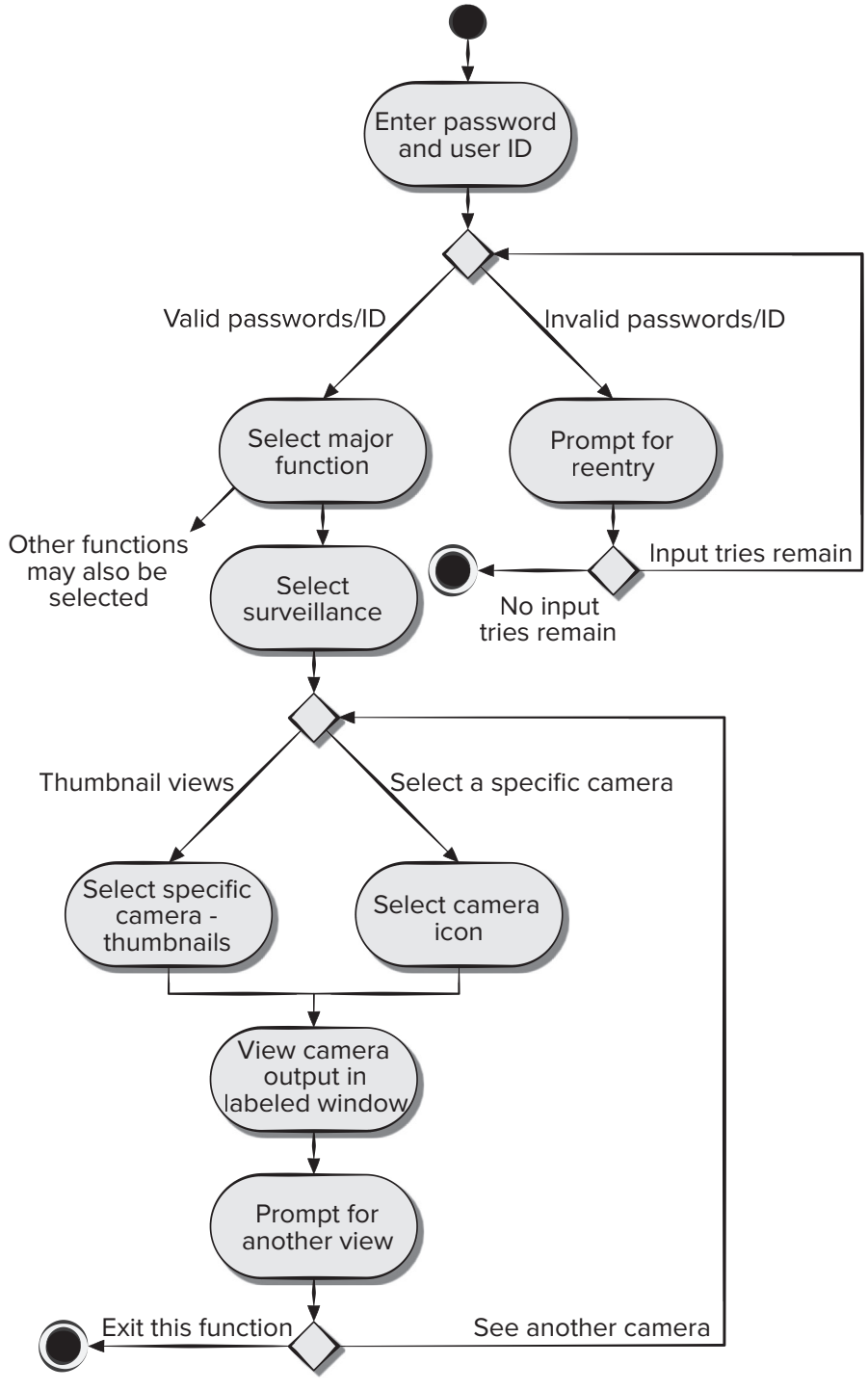
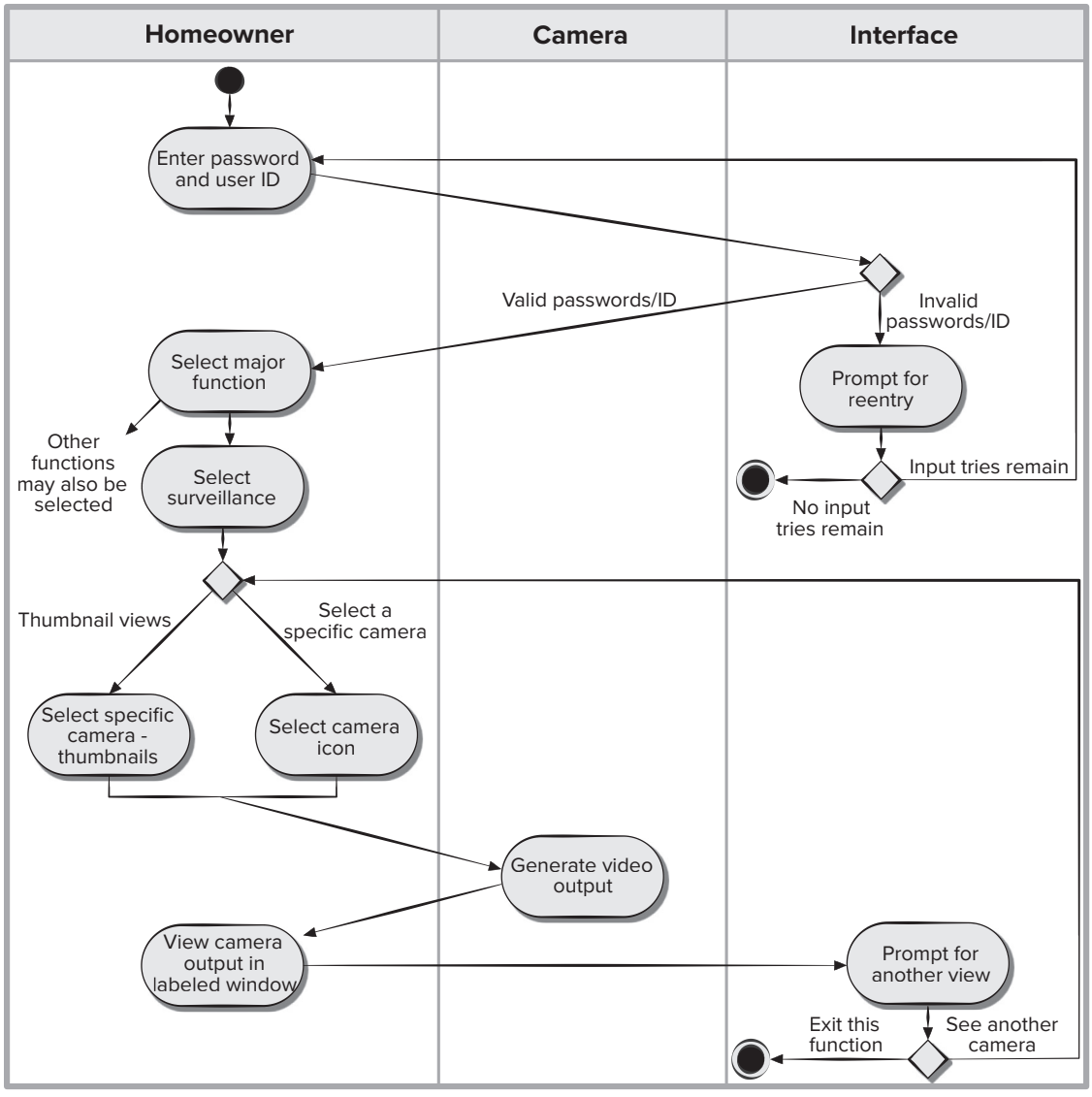


FIGURE 8.10 **Swimlane diagram for Access camera surveillance via the Internet—display camera views function**



Three analysis classes—**Homeowner, Camera,** and **Interface**—have direct or indirect responsibilities in the context of the activity diagram represented in Figure 8.9. Referring to Figure 8.10, the activity diagram is rearranged so that activities associated with an 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.

Use cases, along with the activity and swimlane diagrams, are procedurally oriented. Taken together they can be used to represent the way various actors invoke specific functions (or other procedural steps) to meet the requirements of the system.

8.6 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 description 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 key steps for a specific function or interaction. The degree of use case formality and detail varies, but they can provide necessary input to all other analysis modeling activities. Scenarios can also be described using an activity diagram—a graphical representation that depicts the processing flow within a specific scenario. Temporal relations in a use case can be modeled using sequence diagrams.

Class-based modeling uses information derived from use cases and other written application descriptions 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 using the parse results.

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.

Behavioral modeling during requirements analysis depicts dynamic behavior of the software. The behavioral model uses input from scenario-based or class-based elements to represent the states of analysis classes. To accomplish this, states are identified, the events that cause a class (or the system) to make a transition from one state to another are defined, and the actions that occur as transition is accomplished are also identified. UML state diagrams, activity diagrams, swim lane diagrams, and sequence diagrams can be used for behavioral modeling.

PROBLEMS AND POINTS TO PONDER

8.1. Is it possible to begin coding immediately after a requirements model has been created? Explain your answer, and then argue the counterpoint.

8.2. 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.

8.3. The department of public works for a large city has decided to develop a Web-based pothole tracking and repair system (PHTRS). A description follows:

> Citizens can log onto a website and report the location and severity of potholes. As potholes are reported they are logged within a "public works department repair system" and are assigned an identifying number, stored by street address, size (on a scale of 1 to 10), location (middle, curb, etc.), district (determined from street address), and repair priority (determined from the size of the pothole). Work order data are associated with each pothole and include pothole location and size, repair crew identifying number, number of people on crew, equipment assigned, hours applied to repair, hole status (work in progress, repaired, temporary repair, not repaired), amount of filler material used, and cost of repair (computed from hours applied, number of people, material and equipment used). Finally, a damage file is created to hold information about reported damage due to the pothole and includes citizen's name, address, phone number, type of damage, and dollar amount of damage. PHTRS is an online system; all queries are to be made interactively.

Draw a UML use case diagram PHTRS system. You'll have to make a number of assumptions about the manner in which a user interacts with this system.

8.4. Write two or three use cases that describe the roles of various actors in the PHTRS described in Problem 8.3.

8.5. Develop an activity diagram for one aspect of PHTRS.

8.6. Develop a swimlane diagram for one or more aspects of PHTRS.

8.7. Develop a class model for the PHTRS system presented in Problem 8.3.

8.8. Develop a complete set of CRC model index cards on the product or system you chose as part of Problem 8.3.

8.9. Conduct a review of the CRC index cards with your colleagues. How many additional classes, responsibilities, and collaborators were added as a consequence of the review?

8.10. How does a sequence diagram differ from a state diagram? How are they similar?