CHAPTER

9 Requirements Modeling: Scenario-Based Methods

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

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 a scenario-based perspective and examine how UML can be used to supplement the scenarios. In Chapters 10 and 11, you'll learn about other "dimensions" of the requirements model. By examining a number of different dimensions, you'll increase 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. By building a scenario-based model, you will be able to better understand how the user interacts with the software, uncovering the major functions and features that stakeholder require of the system.

What is the work product? Scenario-based modeling produces a text-oriented representation call a "use case." The use case describes a specific interaction in a manner that can be informal (a simple narrative) or more structured and formal in nature. The use case can be supplemented with a number of different UML diagrams that overlay a more procedural view of the interaction.

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 earlier editions of this book, the term *analysis model* was used, rather than *requirements model*. In this edition, we'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.

particularly to the 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 three decades ago, his comments still apply to modern requirements modeling methods and notation.

9.1 Requirements Analysis

Quote:

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

Alan M. Davis

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 8).

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 the manner in which classes collaborate to achieve system requirements.
- *Behavioral and patterns-based models* that depict how the software behaves as a consequence of 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.

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, we focus on *scenario-based modeling*—a technique that is growing increasingly popular throughout the software engineering community. In Chapters 10 and 11 we consider class-based models and behavioral models. Over the past decade, flow and data modeling have become less commonly used, while scenario and class-based methods, supplemented with behavioral approaches and pattern-based techniques have grown in popularity.[2]

Quote:

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

Andrew Hunt and David Thomas

9.1.1 Overall Objectives and Philosophy

Throughout analysis 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?[3]

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.[4]

KEY POINT

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

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 12 through 18) that describes the software's application architecture, user interface, and component-level structure. This relationship is illustrated in Figure 9.1.

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 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.

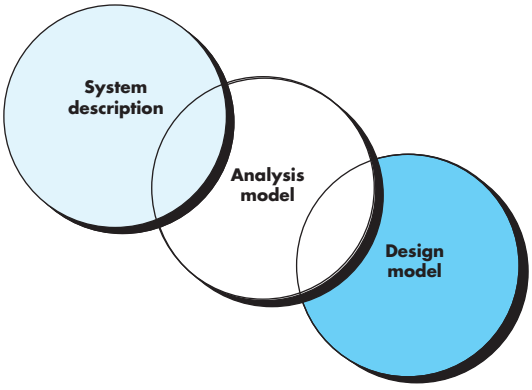
2 Our presentation of flow-oriented modeling and data modeling has been omitted from this edition. However, copious information about these older requirements modeling methods can be found on the Web. If you have interest, use the search phrase "structured analysis."

3 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*.

4 Alternatively, the software team may choose to create a prototype (Chapter 4) in an effort to better understand requirements for the system.

FIGURE 9.1

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



9.1.2 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 some basic guidelines that can guide 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, efforts 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 add 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

9.1.3 Domain Analysis

WebRef

Many useful resources for domain analysis and many other topics can be found at **http://www.sei.cmu.edu/**.

In the discussion of requirements engineering (Chapter 8), we 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.

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:

> 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.

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.

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.[5]

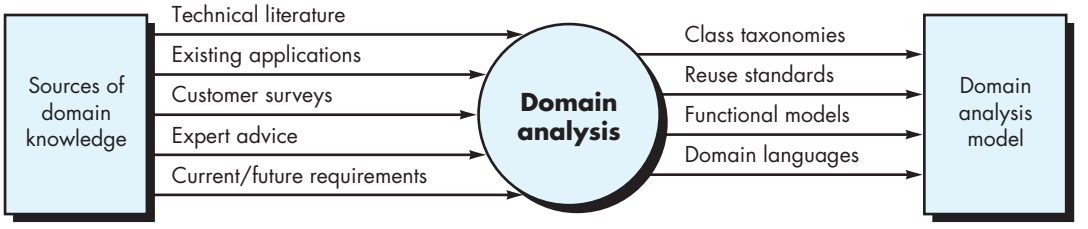
Using terminology that was introduced previously in this book, domain analysis may be viewed as an umbrella activity for the software process. By this we mean that domain analysis is an ongoing software engineering activity that is not connected to any one software project. In a way, the role of a domain 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[6] 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 9.2 [Arn89] 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.

5 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]

6 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.

FIGURE 9.2 Input and output for domain analysis



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.

9.1.4 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

Quote:

"[A]nalysis 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

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 4) are predominantly object oriented.

In this edition of the book, we have chosen to emphasize elements of object-oriented analysis as it is modeled using UML. Our goal is to suggest a combination of representations will provide stakeholders with the best model of software requirements and the most effective bridge to software design.

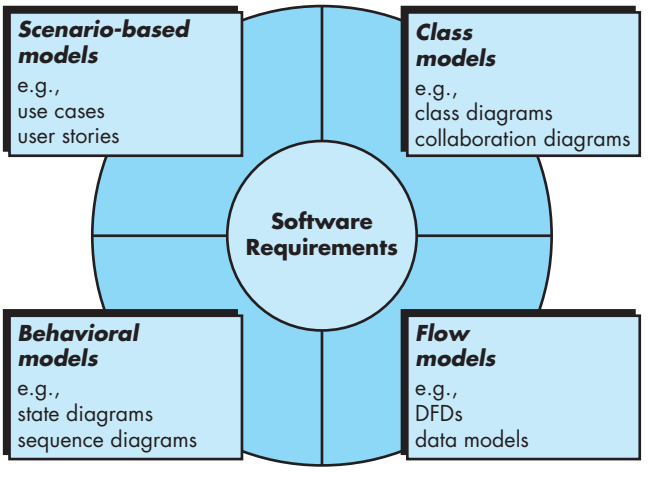
Each element of the requirements model (Figure 9.3) presents the problem from a different point of view. Scenario-based elements depict how the user interacts with the system and the specific sequence of activities that occur as the software is used. Class-based elements model the objects that the system 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.

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

Analysis modeling leads to the derivation of one or more 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.

FIGURE 9.3

Elements of the analysis model



9.2 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[7] begins with the creation of scenarios in the form of use cases, activity diagrams, and swimlane diagrams.

9.2.1 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 8, the "contract" defines the way in which an actor[8] 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, we examine how use cases are developed as part of the analysis modeling activity.[9]

In Chapter 8, 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 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, quality function deployment (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 9.3.1) developed as part of requirements modeling.

7 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 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].

9 Use cases are a particularly important part of analysis modeling for user interfaces. Interface analysis and design is discussed in detail in Chapter 15.

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?

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 8 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.

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

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 more free flowing and did represent a few alternatives). Use cases of this type are sometimes referred to as *primary scenarios* [Sch98a].

9.2.2 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."[10] This error condition becomes a secondary scenario.

10 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.

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.

? What is a use case exception and how do I determine what exceptions are likely?

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 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).

9.2.3 Writing a Formal Use Case

The informal use cases presented in Section 9.2.1 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 9.2.2). 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: Infrequent.

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

Secondary actors: System administrator, cameras.

Channels to secondary actors:

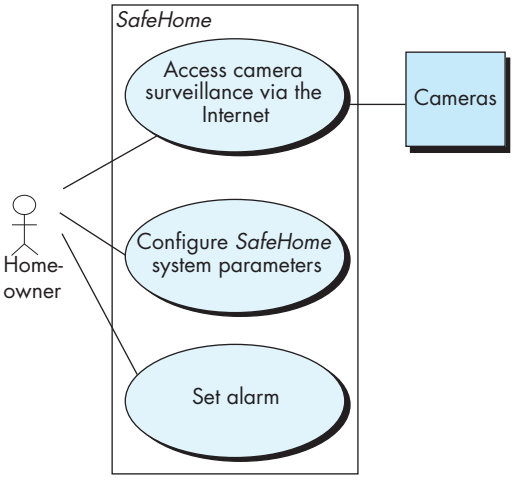
1. System administrator: PC-based system.
2. Cameras: wireless connectivity.

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 9.4

Preliminary use case diagram for the *SafeHome* system



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 9.4 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.

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 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.

9.3 UML MODELS THAT SUPPLEMENT THE USE CASE

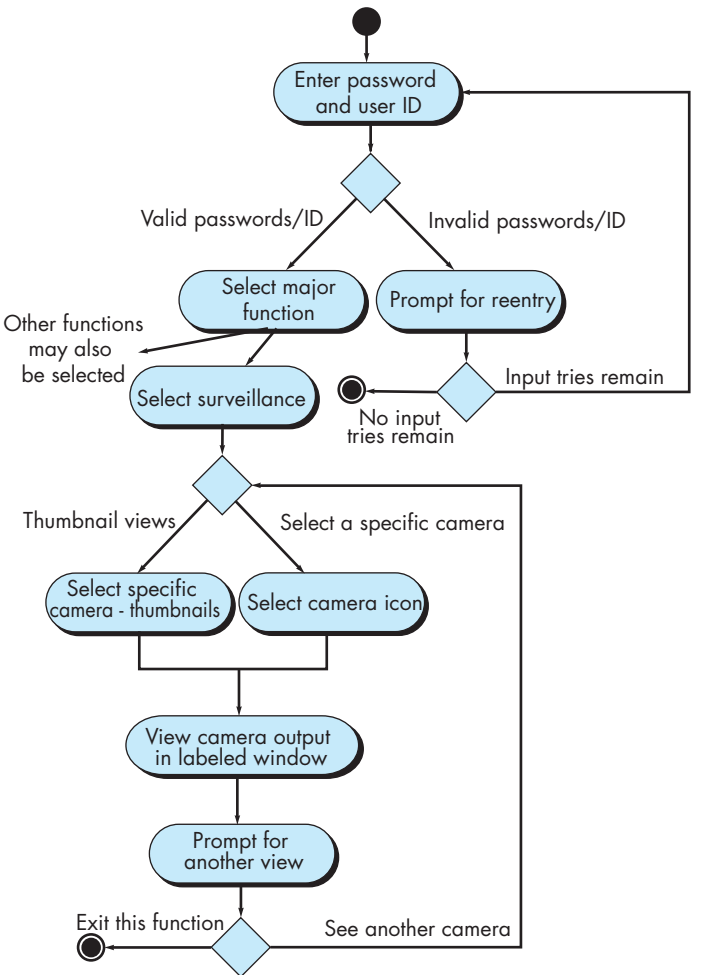
There are many requirements modeling situations in which a text-based model—even one as simple as a use case—may not impart information in a clear and concise manner. In such cases, you can choose from a broad array of UML graphical models.

9.3.1 Developing an Activity Diagram

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 9.5. 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. This is represented by a decision diamond below "Prompt for reentry."

FIGURE 9.5

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



9.3.2 Swimlane Diagrams

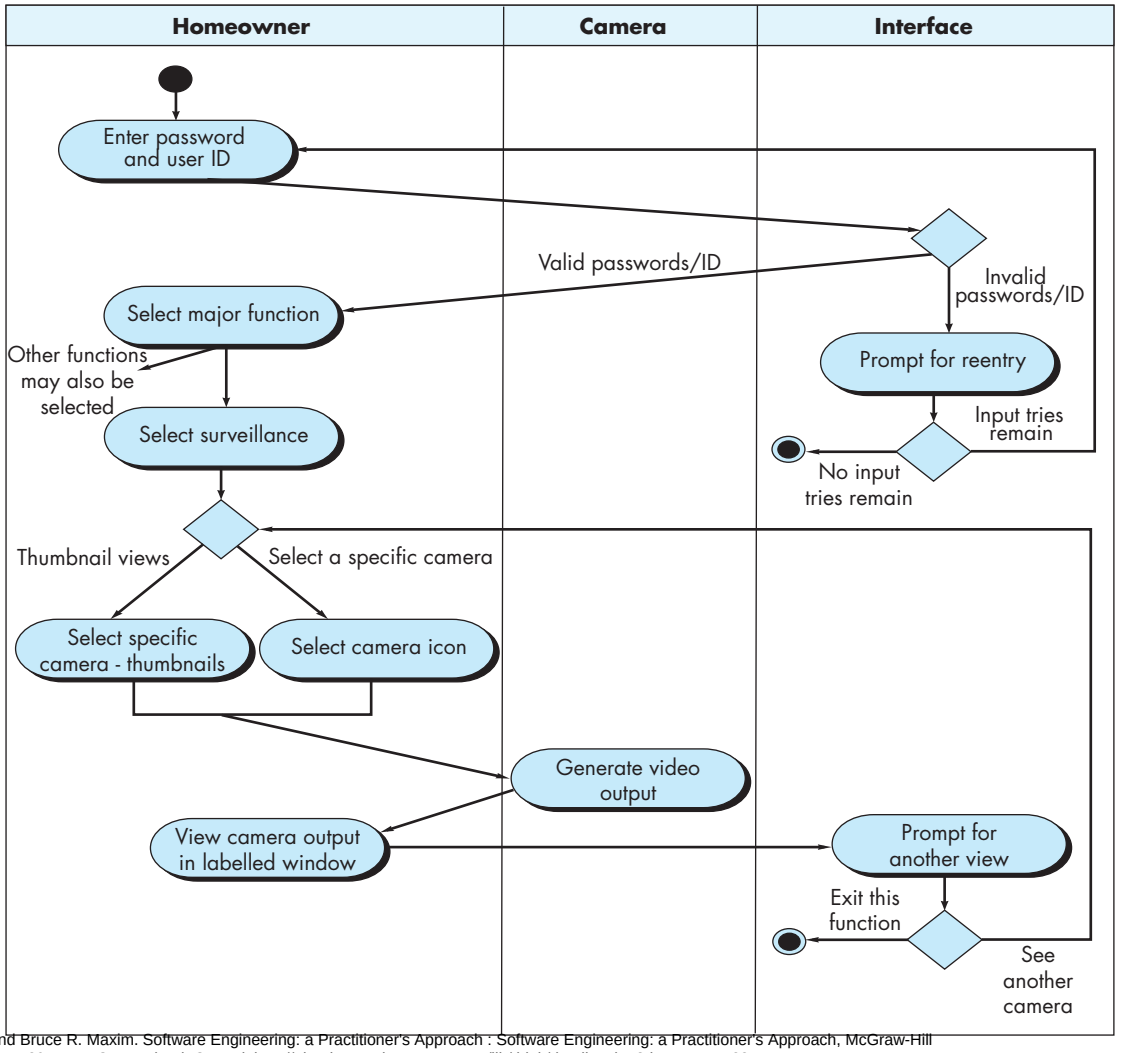
KEY POINT

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 (Chapter 10) 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 9.5. Referring to Figure 9.6, the activity diagram is rearranged so that

FIGURE 9.6 **Swimlane diagram for Access camera surveillance via the Internet—display camera views function.**



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.

Quote:

"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 Chapters 10 and 11, we examine other dimensions of requirements modeling.

9.4 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 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.

Problems and Points to Ponder

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

9.2. 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.