CHAPTER

11 Requirements Modeling: Behavior, Patterns, and Web/Mobile Apps

Key Concepts

After our discussion of scenario-based and class-based models in Chapters 9 and 10, it's reasonable to ask, "Aren't those requirement modeling representations enough?"

The only reasonable answer is, "That depends."

For some types of software, the use case may be the only requirements modeling representation that is required. For others, an object-oriented approach is chosen and class-based models may be developed. But in other situations, complex application requirements may demand an examination of how an application behaves as a consequence of external events; whether existing domain knowledge can be adapted to the current problem; or in the case of Web-based or mobile systems and applications, how content and functionality meld to provide an end user with the ability to successfully navigate an application to achieve usage goals.

Quick Look

What is it? In this chapter you'll learn about other dimensions of the requirements model—behavioral models, patterns, and the special requirements analysis considerations that come into play when WebApps are developed. Each of these modeling representations supplements the scenario-based and class-based models discussed in Chapters 9 and 10.

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

Why is it important? Your insight into software requirements grows in direct proportion to the number of different requirements modeling dimensions. Although you may not have the time, the resources, or the inclination to develop every representation suggested in Chapters 9 to 11, recognize that each different modeling approach provides you with a different way of looking at the problem. As a consequence, you (and other stakeholders) will be better able to assess whether you've properly specified what must be accomplished.

What are the steps? Behavioral modeling depicts the states of the system and its classes and the impact of events on these states. Pattern-based modeling makes use of existing domain knowledge to facilitate requirements analysis. WebApp requirements models are especially adapted for the representation of content, interaction, function, and configuration-related requirements.

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

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

11.1 Creating a Behavioral Model

The modeling notation that has been discussed in the preceding chapters represents static elements of the requirements model. It is now time to make a transition to the dynamic behavior of the system or product. To accomplish this, you can represent the behavior of the system as a function of specific events and time.

? How do I model the software's reaction to some external event?

The *behavioral model* indicates how software will respond to external events or stimuli. 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.

11.2 Identifying Events with the Use Case

In Chapter 9, 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,**[1] transmits an event to the object **ControlPanel.** 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*

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

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).

11.3 State Representations

KEY POINT

The system has states that represent specific externally observable behavior; a class has states that represent its behavior as the system performs its functions.

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 status of all of an object's attributes. For example, the passive state of the class **Player** (in the video game application discussed in Chapter 10) would include the current position and orientation attributes of **Player** as well as other features of **Player** that are relevant to the game (e.g., an attribute that indicates magic wishes remaining). The *active state* of an object indicates the current status of the object as it undergoes a continuing transformation or processing. The class **Player** might have the following active states: *moving, at rest, injured, being cured, trapped, lost,* and so forth. An *event* (sometimes called a *trigger*) must occur to force an object to make a transition from one active state to another.

Two different behavioral representations are discussed in the paragraphs that follow. The first indicates how an individual class changes state based on external events and the second shows the behavior of the software as a function of time.

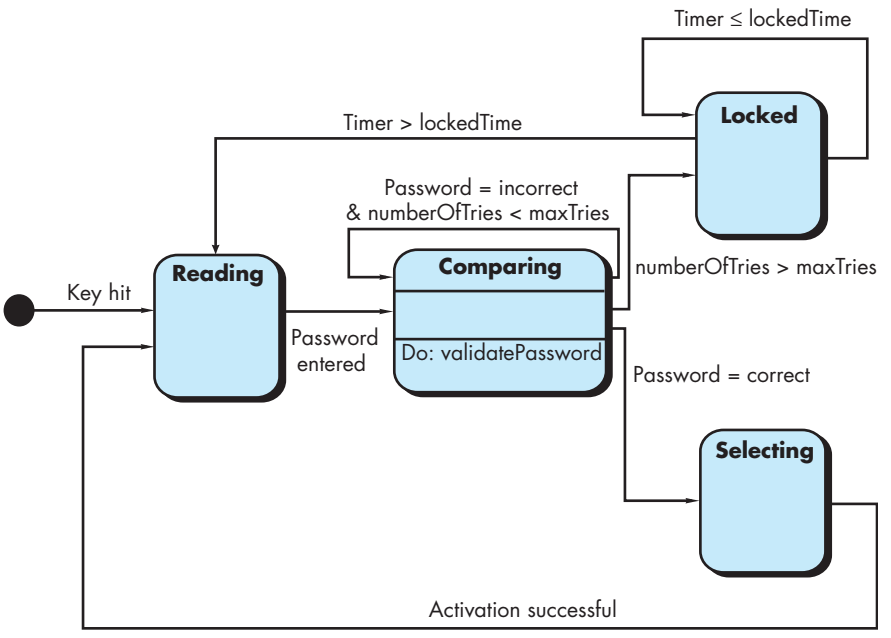
State Diagrams for Analysis Classes. One component of a behavioral model is a UML state diagram[2] that represents active states for each class and the events (triggers) that cause changes between these active states. Figure 11.1 illustrates a state diagram for the **ControlPanel** object in the *SafeHome* security function.

Each arrow shown in Figure 11.1 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

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

FIGURE 11.1

State diagram for the ControlPanel class



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 in order for the transition to occur. For example, the guard for the transition from the "reading" state to the "comparing" state in Figure 11.1 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.

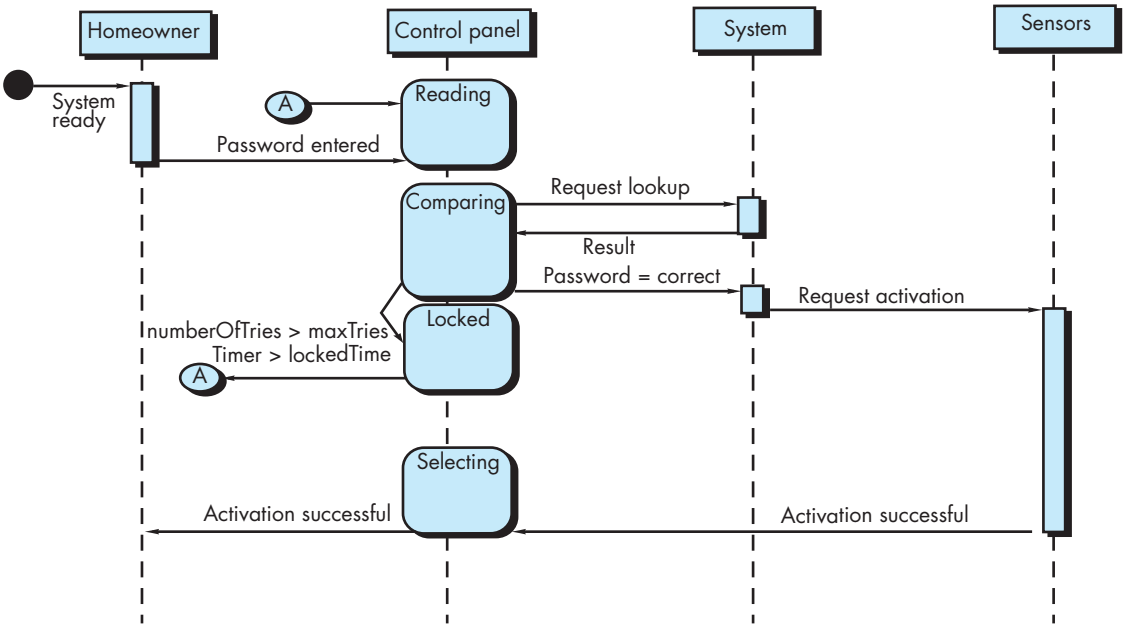
An *action* occurs concurrently with the state transition or as a consequence of it and generally involves one or more operations (responsibilities) of the object. For example, the action connected to the *password entered* event (Figure 11.1) is an operation named *validatePassword()* that accesses a **password** object and performs a digit-by-digit comparison to validate the entered password.

KEY POINT

Unlike a state diagram that represents behavior without noting the classes involved, a sequence diagram represents behavior, by describing how classes move from state to state.

Sequence Diagrams. The second type of behavioral representation, called a *sequence diagram* in UML, indicates 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. In essence, the sequence diagram is

FIGURE 11.2 Sequence diagram (partial) for the *SafeHome* security function



a shorthand version of the use case. It represents key classes and the events that cause behavior to flow from class to class.

Figure 11.2 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.

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 of 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.

SOFTWARE TOOLS

Generalized Analysis Modeling in UML

Objective: Analysis modeling tools provide the capability to develop scenario-based models, class-based models, and behavioral models using UML notation.

Mechanics: Tools in this category support the full range of UML diagrams required to build an analysis model (these tools also support design modeling). In addition to diagramming, tools in this category (1) perform consistency and correctness checks for all UML diagrams, (2) provide links for design and code generation, (3) build a database that enables the management and assessment of large UML models required for complex systems.

Representative Tools:[3]
The following tools support a full range of UML diagrams required for analysis modeling:

ArgoUML is an open source tool available at **argouml.tigris.org**.

Enterprise Architect, developed by Sparx Systems (**www.sparxsystems.com.au**).

PowerDesigner, developed by Sybase (**www.sybase.com**).

Rational Rose, developed by IBM (Rational) (**http://www-01.ibm.com/software/rational/**).

Rational System Architect, developed by Popkin Software now owned by IBM (**http://www-01.ibm.com/software/awdtools/systemarchitect/**).

UML Studio, developed by Pragsoft Corporation (**www.pragsoft.com**).

Visio, developed by Microsoft (**http://office.microsoft.com/en-gb/visio/**).

Visual UML, developed by Visual Object Modelers (**www.visualuml.com**).

11.4 PATTERNS FOR REQUIREMENTS MODELING

Software patterns are a mechanism for capturing domain knowledge in a way that allows it to be reapplied when a new problem is encountered. In some cases, the domain knowledge is applied to a new problem within the same application domain. In other cases, the domain knowledge captured by a pattern can be applied by analogy to a completely different application domain.

The original author of an analysis pattern does not "create" the pattern, but, rather, *discovers* it as requirements engineering work is being conducted. Once the pattern has been discovered, it is documented by describing "explicitly the general problem to which the pattern is applicable, the prescribed solution, assumptions and constraints of using the pattern in practice, and often some other information about the pattern, such as the motivation and driving forces for using the pattern, discussion of the pattern's advantages and disadvantages, and references to some known examples of using that pattern in practical applications" [Dev01].

In Chapter 8, we introduced the concept of analysis patterns and indicated that these patterns represent something (e.g., a class, a function, a behavior)

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

within the application domain that can be reused when performing requirements modeling for an application within a domain.[4] Analysis patterns are stored in a repository so that members of the software team can use search facilities to find and reuse them. Once an appropriate pattern is selected, it is integrated into the requirements model by reference to the pattern name.

11.4.1 Discovering Analysis Patterns

The requirements model comprises a wide variety of elements: scenario-based (use cases), class-based (objects and classes), and behavioral (events and states). Each of these elements represents the problem from a different perspective, and each provides an opportunity to discover patterns that may occur throughout an application domain, or by analogy, across different application domains.

The most basic element in the description of a requirements model is the use case. In the context of this discussion, a coherent set of use cases may serve as the basis for discovering one or more analysis patterns. A *semantic analysis pattern* (SAP) "is a pattern that describes a small set of coherent use cases that together describe a basic generic application" [Fer00].

Consider the following preliminary use case for software required to control and monitor a real-view camera and proximity sensor for an automobile:

> **Use case:** *Monitor reverse motion*
>
> **Description:** When the vehicle is placed in *reverse* gear, the control software enables a video feed from a rear-placed video camera to the dashboard display. The control software superimposes a variety of distance and orientation lines on the dashboard display so that the vehicle operator can maintain orientation as the vehicle moves in reverse. The control software also monitors a proximity sensor to determine whether an object is inside 10 feet of the rear of the vehicle. It will automatically brake the vehicle if the proximity sensor indicates an object within x feet of the rear of the vehicle, where x is determined based on the speed of the vehicle.

This use case implies a variety of functionality that would be refined and elaborated (into a coherent set of use cases) during requirements gathering and modeling. Regardless of how much elaboration is accomplished, the use cases suggest a simple, yet widely applicable SAP—the software-based monitoring and control of sensors and actuators in a physical system. In this case, the "sensors" provide information about proximity and video information. The "actuator" is the braking system of the vehicle (invoked if an object is close to the vehicle). But in a more general case, a widely applicable pattern is discovered.

4 An in-depth discussion of the use of patterns during software design is presented in Chapter 16.

SafeHome

Discovering an Analysis Pattern

The scene: A meeting room, during a team meeting.

The players: Jamie Lazar, software team member; Ed Robbins, software team member; Doug Miller, software engineering manager

The conversation:

Doug: How are things going with modeling the requirements for the sensor network for the *SafeHome* project?

Jamie: Sensor work is a little new to me, but I think I'm getting a handle on it.

Doug: Is there anything we can do to help you with that?

Jamie: It would be a lot easier if I'd built a system like this before.

Doug: True.

Ed: I was thinking this is a situation where we might be able to find an analysis pattern that would help us model these requirements.

Doug: If we can find the right pattern, we'd avoid reinventing the wheel.

Jamie: That sounds good to me. How do we start?

Ed: We have access to a repository that contains a large number of analysis and design patterns. We just need to search for patterns with intents that match our needs.

Doug: That seems like that might work. What do you think, Jamie?

Jamie: If Ed can help me get started, I'll tackle this today.

Software in many different application domains is required to monitor sensors and control physical actuators. It follows that an analysis pattern that describes generic requirements for this capability could be used widely. The pattern, called **Actuator-Sensor**, would be applicable as part of the requirements model for *SafeHome* and is discussed in Section 11.4.2.

11.4.2 A Requirements Pattern Example: Actuator-Sensor[5]

One of the requirements of the *SafeHome* security function is the ability to monitory security sensors (e.g., break-in sensors, fire, smoke or CO sensors, water sensors). Internet-based extensions to *SafeHome* will require the ability to control the movement (e.g., pan, zoom) of a security camera within a residence. The implication—*SafeHome* software must manage various sensors and "actuators" (e.g., camera control mechanisms).

Konrad and Cheng [Kon02] have suggested a requirements pattern named **Actuator-Sensor** that provides useful guidance for modeling this requirement within *SafeHome* software. An abbreviated version of the **Actuator-Sensor** pattern, originally developed for automotive applications, follows.

Pattern Name. **Actuator-Sensor**

Intent. Specify various kinds of sensors and actuators in an embedded system.

5 This section has been adapted from [Kon02] with the permission of the authors.

Motivation. Embedded systems usually have various kinds of sensors and actuators. These sensors and actuators are all either directly or indirectly connected to a control unit. Although many of the sensors and actuators look quite different, their behavior is similar enough to structure them into a pattern. The pattern shows how to specify the sensors and actuators for a system, including attributes and operations. The **Actuator-Sensor** pattern uses a *pull* mechanism (explicit request for information) for **PassiveSensors** and a *push* mechanism (broadcast of information) for the **ActiveSensors.**

Constraints

- Each passive sensor must have some method to read sensor input and attributes that represent the sensor value.
- Each active sensor must have capabilities to broadcast update messages when its value changes.
- Each active sensor should send a *life tick*, a status message issued within a specified time frame, to detect malfunctions.
- Each actuator must have some method to invoke the appropriate response determined by the **ComputingComponent.**
- Each sensor and actuator should have a function implemented to check its own operation state.
- Each sensor and actuator should be able to test the validity of the values received or sent and set its operation state if the values are outside of the specifications.

Applicability. Useful in any system in which multiple sensors and actuators are present.

Structure. UML class diagram for the **Actuator-Sensor** pattern is shown in Figure 11.3. **Actuator, PassiveSensor,** and **ActiveSensor** are abstract classes and denoted in italics. There are four different types of sensors and actuators in this pattern. The **Boolean, Integer,** and **Real** classes represent the most common types of sensors and actuators. The complex classes are sensors or actuators that use values that cannot be easily represented in terms of primitive data types, such as a radar device. Nonetheless, these devices should still inherit the interface from the abstract classes since they should have basic functionalities such as querying the operation states.

Behavior. Figure 11.4 presents a UML sequence diagram for an example of the **Actuator-Sensor** pattern as it might be applied for the *SafeHome* function that controls the positioning (e.g., pan, zoom) of a security camera. Here, the **ControlPanel**[6] queries a sensor (a passive position sensor) and an actuator (pan

6 The original pattern uses the generic phrase **ComputingComponent.**

FIGURE 11.3

UML sequence diagram for the Actuator-Sensor pattern. *Source:* Adapted from [Kon02] with permission.

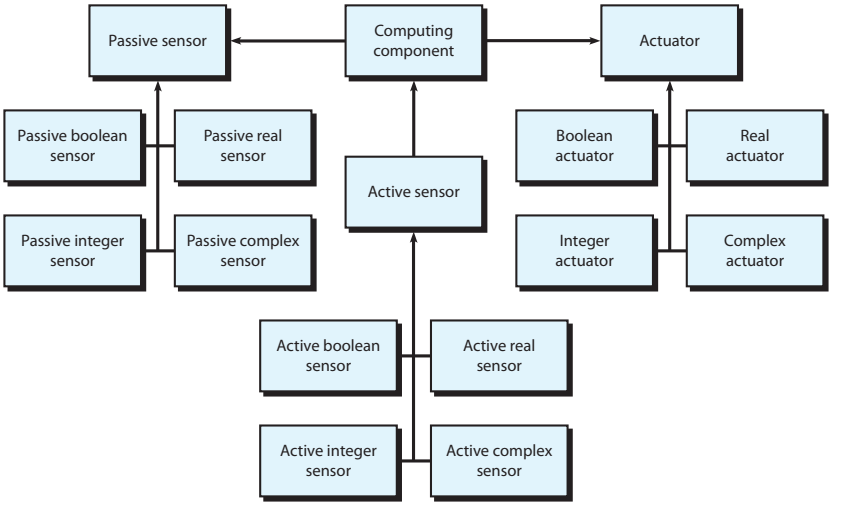
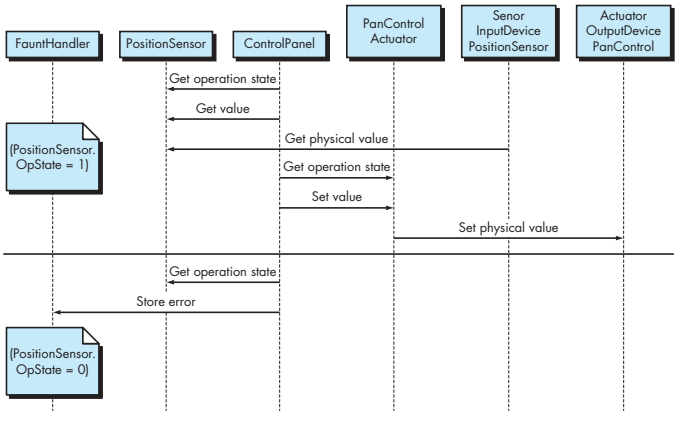


FIGURE 11.4

UML Class diagram for the Actuator-Sensor pattern. *Source:* Reprinted from [Kon02] with permission.



control) to check the operation state for diagnostic purposes before reading or setting a value. The messages *Set Physical Value* and *Get Physical Value* are not messages between objects. Instead, they describe the interaction between the physical devices of the system and their software counterparts. In the lower part of the diagram, below the horizontal line, the **PositionSensor** reports that the operation state is zero. The **ComputingComponent** then sends the error code for a position sensor failure to the **FaultHandler** that will decide how this error affects the system and what actions are required. It gets the data from the sensors and computes the required response for the actuators.

Participants. This section of the patterns description "itemizes the classes/objects that are included in the requirements pattern" [Kon02] and describes the responsibilities of each class/object (Figure 11.3). An abbreviated list follows:

- **PassiveSensor abstract:** Defines an interface for passive sensors.
- **PassiveBooleanSensor:** Defines passive Boolean sensors.
- **PassiveIntegerSensor:** Defines passive integer sensors.
- **PassiveRealSensor:** Defines passive real sensors.
- **ActiveSensor abstract:** Defines an interface for active sensors.
- **ActiveBooleanSensor:** Defines active Boolean sensors.
- **ActiveIntegerSensor:** Defines active integer sensors.
- **ActiveRealSensor:** Defines active real sensors.
- **Actuator abstract:** Defines an interface for actuators.
- **BooleanActuator:** Defines Boolean actuators.
- **IntegerActuator:** Defines integer actuators.
- **RealActuator:** Defines real actuators.
- **ComputingComponent:** The central part of the controller; it gets the data from the sensors and computes the required response for the actuators.
- **ActiveComplexSensor:** Complex active sensors have the basic functionality of the abstract **ActiveSensor** class, but additional, more elaborate, methods and attributes need to be specified.
- **PassiveComplexSensor:** Complex passive sensors have the basic functionality of the abstract **PassiveSensor** class, but additional, more elaborate, methods and attributes need to be specified.
- **ComplexActuator:** Complex actuators also have the base functionality of the abstract **Actuator** class, but additional, more elaborate methods and attributes need to be specified.

Collaborations. This section describes how objects and classes interact with one another and how each carries out its responsibilities.

- When the **ComputingComponent** needs to update the value of a **PassiveSensor,** it queries the sensors, requesting the value by sending the appropriate message.
- **ActiveSensors** are not queried. They initiate the transmission of sensor values to the computing unit, using the appropriate method to set the value in the **ComputingComponent.** They send a life tick at least once during a specified time frame in order to update their timestamps with the system clock's time.

- When the **ComputingComponent** needs to set the value of an actuator, it sends the value to the actuator.
- The **ComputingComponent** can query and set the operation state of the sensors and actuators using the appropriate methods. If an operation state is found to be zero, then the error is sent to the **FaultHandler,** a class that contains methods for handling error messages, such as starting a more elaborate recovery mechanism or a backup device. If no recovery is possible, then the system can only use the last known value for the sensor or the default value.
- The **ActiveSensors** offer methods to add or remove the addresses or address ranges of the components that want to receive the messages in case of a value change.

Consequences

1. Sensor and actuator classes have a common interface.
2. Class attributes can only be accessed through messages, and the class decides whether or not to accept the message. For example, if a value of an actuator is set above a maximum value, then the actuator class may not accept the message, or it might use a default maximum value.
3. The complexity of the system is potentially reduced because of the uniformity of interfaces for actuators and sensors.

The requirements pattern description might also provide references to other related requirements and design patterns.

11.5 Requirements Modeling for Web and Mobile Apps[7]

Developers of Web and mobile applications are often skeptical when the idea of requirements analysis is suggested. "After all," they argue, "our development process must be agile, and analysis is time consuming. It'll slow us down just when we need to be designing and building the application."

Requirements analysis does take time, but solving the wrong problem takes even more time. The question for every WebApp and mobile developer is simple—are you sure you understand the requirements of the problem or product? If the answer is an unequivocal yes, then it may be possible to skip requirements modeling, but if the answer is no, then requirements modeling should be performed.

7 Portions of this section has been adapted from Pressman and Lowe [Pre08] with permission.

11.5.1 How Much Analysis Is Enough?

The degree to which requirements modeling for Web and mobile apps is emphasized depends on the following size-related factors: (1) the size and complexity of the application increment, (2) the number of stakeholders (analysis can help to identify conflicting requirements coming from different sources), (3) the size of the app development team, (4) the degree to which members of the team have worked together before (analysis can help develop a common understanding of the project), and (5) the degree to which the organization's success is directly dependent on the success of the application.

The converse of the preceding points is that as the project becomes smaller, the number of stakeholders fewer, the development team more cohesive, and the application less critical, it is reasonable to apply a more lightweight analysis approach.

Although it is a good idea to analyze the problem or product requirements *before* beginning design, it is not true that *all* analysis must precede *all* design. In fact, the design of a specific part of the application only demands an analysis of those requirements that affect only that part of the application. As an example from *SafeHome*, you could validly design the overall website aesthetics (layouts, color schemes, etc.) without having analyzed the functional requirements for e-commerce capabilities. You only need to analyze that part of the problem that is relevant to the design work for the increment to be delivered.[8]

11.5.2 Requirements Modeling Input

An agile version of the generic software process discussed in Chapter 5 can be applied when Web or mobile apps are engineered. The process incorporates a communication activity that identifies stakeholders and user categories, the business context, defined informational and applicative goals, general product requirements, and usage scenarios—information that becomes input to requirements modeling. This information is represented in the form of natural language descriptions, rough outlines, sketches, and other informal representations.

Analysis takes this information, structures it using a formally defined representation scheme (where appropriate), and then produces more rigorous models as an output. The requirements model provides a detailed indication of the true structure of the problem and provides insight into the shape of the solution.

The *SafeHome* ACS-DCV (camera surveillance) function was introduced in Chapter 9. When it was introduced, this function seemed relatively clear and was described in some detail as part of a use case (Section 9.2.1). However, a reexamination of the use case might uncover information that is missing, ambiguous, or unclear.

Some aspects of this missing information would naturally emerge during the design. Examples might include the specific layout of the function buttons, their

8 In situations in which a design of one part of an application will have impact across other parts of an application, the scope of analysis should be broadened.

aesthetic look and feel, the size of snapshot views, the placement of camera views and the house floor plan, or even minutiae such as the maximum and minimum length of passwords. Some of these aspects are design decisions (such as the layout of the buttons) and others are requirements (such as the length of the passwords) that don't fundamentally influence early design work.

But some missing information might actually influence the overall design itself and relate more to an actual understanding of the requirements. For example:

Q1: What output video resolution is provided by *SafeHome* cameras?

Q2: What occurs if an alarm condition is encountered while the camera is being monitored?

Q3: How does the system handle cameras that can be panned and zoomed?

Q4: What information should be provided along with the camera view? (For example, location? time/date? last previous access?)

None of these questions were identified or considered in the initial development of the use case, and yet, the answers could have a substantial effect on different aspects of the design.

Therefore, it is reasonable to conclude that although the communication activity provides a good foundation for understanding, requirements analysis refines this understanding by providing additional interpretation. As the problem structure is delineated as part of the requirements model, questions invariably arise. It is these questions that fill in the gaps—or in some cases, actually help us to find the gaps in the first place.

To summarize, the inputs to the requirements model will be the information collected during the communication activity—anything from an informal e-mail to a detailed project brief complete with comprehensive usage scenarios and product specifications.

11.5.3 Requirements Modeling Output

Requirements analysis provides a disciplined mechanism for representing and evaluating application content and function, the modes of interaction that users will encounter, and the environment and infrastructure in which the WebApp or mobile app resides.

Each of these characteristics can be represented as a set of models that allow application requirements to be analyzed in a structured manner. While the specific models depend largely upon the nature of the application, there are five main classes of models:

- **Content model**—identifies the full spectrum of content to be provided by the application. Content includes text, graphics and images, video, and audio data.

- **Interaction model**—describes the manner in which users interact with the app.
- **Functional model**—defines the operations that will be applied to manipulate content and describes other processing functions that are independent of content but necessary to the end user.
- **Navigation model**—defines the overall navigation strategy for the app.
- **Configuration model**—describes the environment and infrastructure in which the app resides.

You can develop each of these models using a representation scheme (often called a "language") that allows its intent and structure to be communicated and evaluated easily among members of the engineering team and other stakeholders. As a consequence, a list of key issues (e.g., errors, omissions, inconsistencies, suggestions for enhancement or modification, points of clarification) are identified and acted upon.

11.5.4 Content Model

The content model contains structural elements that provide an important view of content requirements for an application. These structural elements encompass content objects and all *analysis classes*—user-visible entities that are created or manipulated as a user interacts with the app through a browser or a mobile device.[9]

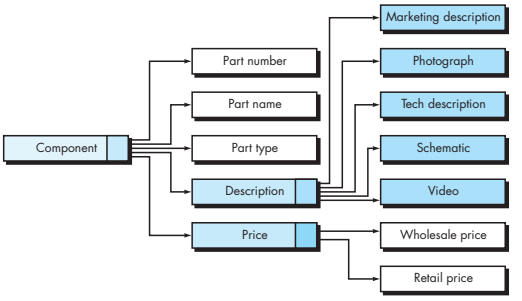
Content can be developed prior to the implementation of the app, while the app is being built, or long after the app is operational. In every case, it is incorporated via navigational reference into the overall application structure. A *content object* might be a textual description of a product, an article describing a news event, a graphical representation of retrieved data (e.g., stock price as a function of time), an action photograph taken at a sporting event, a user's response on a discussion forum, an animated representation of a corporate logo, a short video of a speech, or an audio overlay for a collection of presentation slides. The content objects might be stored as separate files or obtained dynamically from a database. They might be embedded directly into Web pages, displayed on the screen of a mobile device. In other words, a content object is any item of cohesive information that is to be presented to an end user.

Content objects can be determined directly from use cases by examining the scenario description for direct and indirect references to content. For example, a WebApp that supports *SafeHome* is established at **www.safehomeassured.com.** A use case, *Purchasing Select SafeHome Components,* describes the scenario required to purchase a *SafeHome* component and contains the sentence:

> I will be able to get descriptive and pricing information for each product component.

9 Analysis classes were discussed in Chapter 10.

FIGURE 11.5 Data tree for a **www.safehomeassured.com** component



The content model must be capable of describing the content object **Component.** In many instances, a simple list of content objects, coupled with a brief description of each object, is sufficient to define the requirements for content that must be designed and implemented. However, in some cases, the content model may benefit from a richer analysis that graphically illustrates the relationships among content objects and/or the hierarchy of content maintained by a WebApp.

For example, consider the *data tree* [Sri01] created for a **www.safehomeassured.com** component shown in Figure 11.5. The tree represents a hierarchy of information that is used to describe a component. Simple or composite data items (one or more data values) are represented as unshaded rectangles. Content objects are represented as shaded rectangles. In the figure, description is defined by five content objects (the shaded rectangles). In some cases, one or more of these objects would be further refined as the data tree expands.

A data tree can be created for any content that is composed of multiple content objects and data items. The data tree is developed in an effort to define hierarchical relationships among content objects and to provide a means for reviewing content so that omissions and inconsistencies are uncovered before design commences. In addition, the data tree serves as the basis for content design.

11.5.5 Interaction Model for Web and Mobile Apps

The vast majority of Web and mobile apps enable a "conversation" between an end user and application functionality, content, and behavior. This conversation can be described using an *interaction* model that can be composed of one or more of the following elements: (1) use cases, (2) sequence diagrams, (3) state diagrams,[10] and/or (4) user interface prototypes.

In many instances, a set of use cases[11] is sufficient to describe the interaction at an analysis level (further refinement and detail is introduced during design).

10 Sequence diagrams and state diagrams are modeled using UML notation.

11 Use cases are described in detail in Chapter 9.

However, when the sequence of interaction is complex and involves multiple analysis classes or many tasks, it is sometimes worthwhile to depict it using a more rigorous diagrammatic form.

The layout of the user interface, the content it presents, the interaction mechanisms it implements, and the overall aesthetic of the user to app connection have much to do with user satisfaction and the overall success of the app. Although it can be argued that the creation of a user interface prototype is a design activity, it is a good idea to perform it during the creation of the analysis model. The sooner that a physical representation of a user interface can be reviewed, the higher the likelihood that end users will get what they want. The design of user interfaces is discussed in detail in Chapter 15.

Because Web and mobile app construction tools are plentiful, relatively inexpensive, and functionally powerful, it is best to create the interface prototype using such tools. The prototype should implement the major navigational links and represent the overall screen layout in much the same way that it will be constructed. For example, if five major system functions are to be provided to the end user, the prototype should represent them as the user will see them upon first entering the app. Will graphical links be provided? Where will the navigation menu be displayed? What other information will the user see? Questions like these should be answered by the prototype.

11.5.6 Functional Model

Many WebApps deliver a broad array of computational and manipulative functions that can be associated directly with content (either using it or producing it) and that are often a major goal of user-WebApp interaction. Mobile apps tend to be more focused and provide a more limited set of computational and manipulative functions. Regardless of the breadth of functionality, functional requirements should be analyzed, and when necessary, modeled.

The *functional model* addresses two app 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 actually 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.

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 WebApps and 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.

An example of relatively complex functionality for **www.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. **www.safehomeassured.com** responds with a graphical representation of the layout with additional information on the recommended locations for sensors. The interaction is quite simple, the content is somewhat more complex, but the underlying functionality it very sophisticated. The system must undertake a relatively complex analysis of the floor layout in order to determine the optimal set of sensors. It must examine room dimensions, 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.

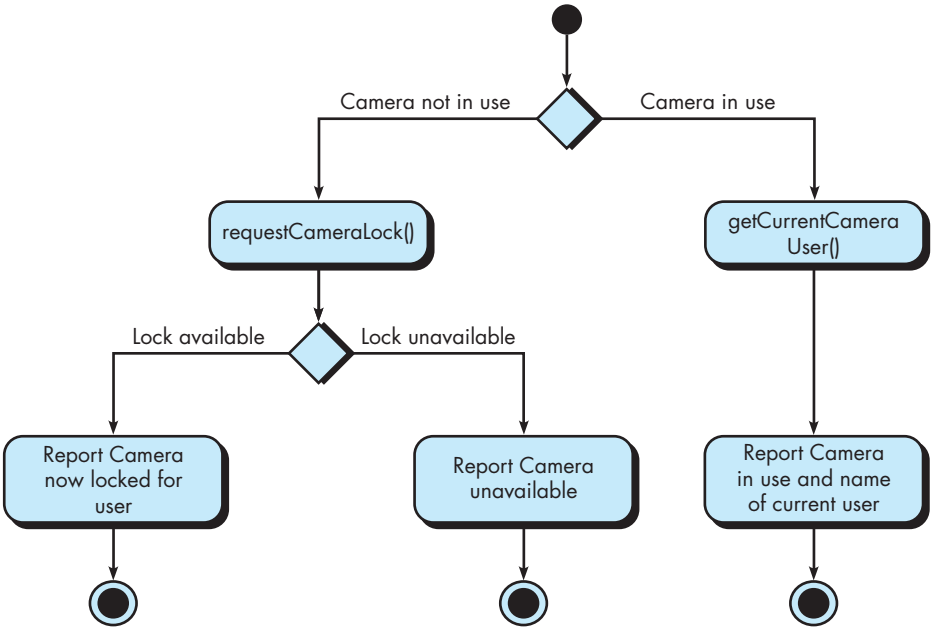
Figure 11.6 depicts an activity diagram for the *takeControlOfCamera()* operation that is part of the **Camera** analysis class used within the *Control cameras* use case. It should be noted that two additional operations are invoked with the procedural flow: *requestCameraLock(),* which tries to lock 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 WebApp design commences.

An extension of *SafeHome* WebApp functionality might occur with the development of a mobile app that provides access to the *SafeHome* system from a smart phone or tablet. The content and functional requirements for a *SafeHome* mobile app might be similar to a subset of those provided by the WebApp, but specific interface and security requirements would have to be established.

11.5.7 Configuration Models for WebApps

In some cases, the configuration model is nothing more than a list of server-side and client-side attributes. However, for more complex apps, a variety of configuration

FIGURE 11.6
Activity diagram for the *takeControlOf-Camera()* operation



complexities (e.g., distributing load among multiple servers, caching architectures, remote databases, multiple servers serving various objects) may have an impact on analysis and design. The UML *deployment diagram* can be used in situations in which complex configuration architectures must be considered.

For **www.safehomeassured.com** the public content and functionality should be specified to be accessible across all major Web clients (i.e., those with more than 1 percent market share or greater). Conversely, it may be acceptable to restrict the more complex control and monitoring functionality (which is only accessible to **HomeOwner** users) to a smaller set of clients. For a mobile app, implementation might be limited to the three leading mobile operating environments. The configuration model for **www.safehomeassured.com** will also specify interoperability with existing product databases and monitoring applications.

11.5.8 Navigation Modeling

In most mobile applications that reside on smartphone platforms, navigation is generally constrained to relatively simple button lists and icon-based menus. In addition, the depth of navigation (i.e., the number of levels into the hypermedia hierarchy) is relatively shallow. For these reasons, navigation modeling is relatively simple.

For WebApps and an increasing number of tablet-based mobile applications, navigation modeling is more complex and often considers how each user category will navigate from one WebApp element (e.g., content object) to another. The mechanics

of navigation are defined as part of design. At this stage, you should focus on overall navigation requirements. The following questions should be considered:

- Should certain elements be easier to reach (require fewer navigation steps) than others? What is the priority for presentation?
- Should certain elements be emphasized to force users to navigate in their direction?
- How should navigation errors be handled?
- Should navigation to related groups of elements be given priority over navigation to a specific element?
- Should navigation be accomplished via links, via search-based access, or by some other means?
- Should certain elements be presented to users based on the context of previous navigation actions?
- Should a navigation log be maintained for users?
- Should a full navigation map or menu (as opposed to a single "back" link or directed pointer) be available at every point in a user's interaction?
- Should navigation design be driven by the most commonly expected user behaviors or by the perceived importance of the defined WebApp elements?
- Can a user "store" his previous navigation through the WebApp to expedite future usage?
- For which user category should optimal navigation be designed?
- How should links external to the WebApp be handled? Overlaying the existing browser window? As a new browser window? As a separate frame?

These and many other questions should be asked and answered as part of navigation analysis.

You and other stakeholders must also determine overall requirements for navigation. For example, will a "site map" be provided to give users an overview of the entire WebApp structure? Can a user take a "guided tour" that will highlight the most important elements (content objects and functions) that are available? Will a user be able to access content objects or functions based on defined attributes of those elements (e.g., a user might want to access all photographs of a specific building or all functions that allow computation of weight)?

11.6 Summary

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 and the system as a