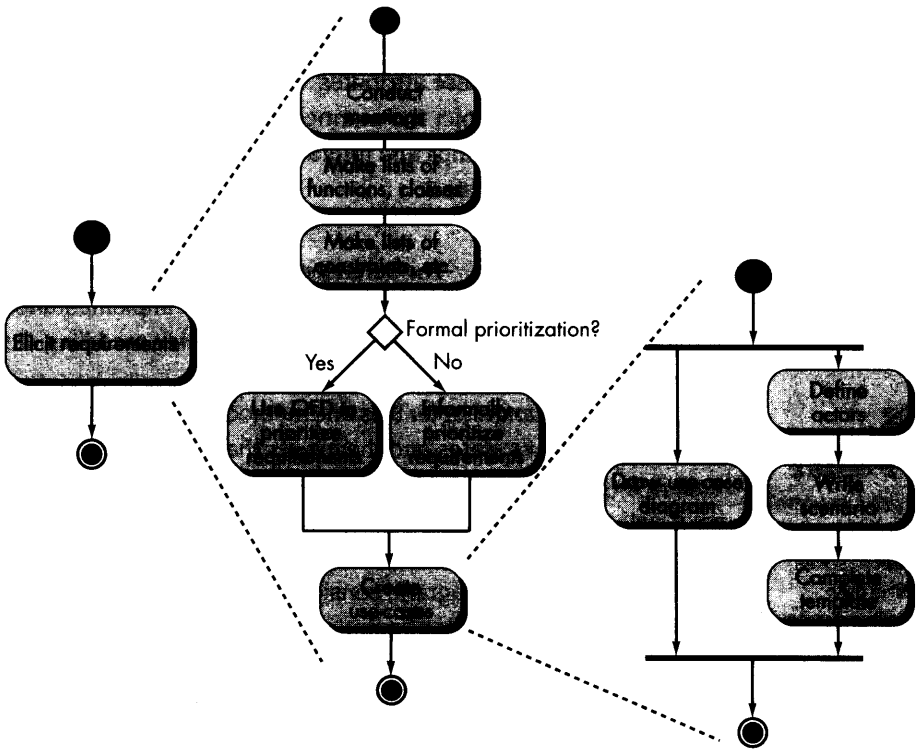
It always a good idea to get stakeholders involved. One of the best ways to do this is to have each stakeholder write use-cases that describe how the software will be used.

Scenario-based elements. The system is described from the user's point of view using a scenario-based approach. For example, basic use-cases (Section 7.5) and their corresponding use-case diagrams (Figure 7.3) evolve into more elaborate template-based use-cases. Scenario-based elements of the analysis model are often the first part of the analysis model that is developed. As such, they serve as input for the creation of other modeling elements.

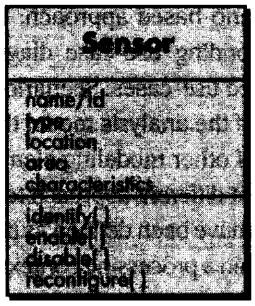
A somewhat different approach to scenario-based modeling depicts the activities or functions that have been defined as part of the requirement elicitation task. These functions exist within a processing context. That is, the sequence of activities (the terms *functions* or *operations* can also be used) that describe processing within a limited context are defined as part of the analysis model. Like most elements of the analysis model (and other software engineering models), activities (functions) can be represented at many different levels of abstraction. Models in this category can be defined iteratively. Each iteration provides additional processing detail. As an example, Figure 7.4 depicts a UML activity diagram for eliciting requirements.[14] Three levels of elaboration are shown.

FIGURE 7.4
Activity diagrams for eliciting requirements



14 The activity diagram is quite similar to the flowchart—a graphical diagram for representing control-flow sequences and logic (Chapter 11).

FIGURE 7.5
Class diagram for Sensor



One way to isolate classes is to look for descriptive nouns in a use-case script. At least some of the nouns will be candidate classes. More on this in Chapter 8.

Class-based elements. Each usage scenario implies a set of "objects" that are manipulated as an actor interacts with the system. These objects are categorized into classes—a collection of things that have similar attributes and common behaviors. For example, a class diagram can be used to depict a **Sensor** class for the *SafeHome* security function (Figure 7.5). Note that the diagram lists the attributes of sensors (e.g., **name/id, type**) and the operations [e.g., *identify(), enable()*] that can be applied to modify these attributes. In addition to class diagrams, other analysis modeling elements depict the manner in which classes collaborate with one another and the relationships and interactions between classes. These are discussed in more detail in Chapter 8.

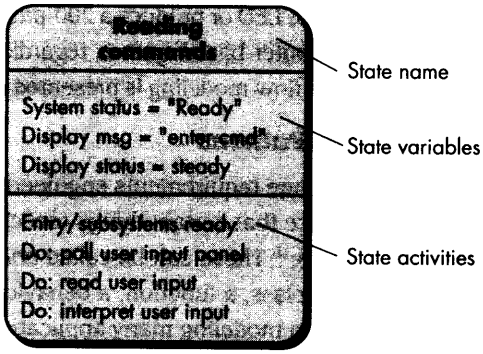
Behavioral elements. The behavior of a computer-based system can have a profound effect on the design that is chosen and the implementation approach that is applied. Therefore, the analysis model must provide modeling elements that depict behavior.

A state is an externally observable mode of behavior. External stimuli cause transitions between states.

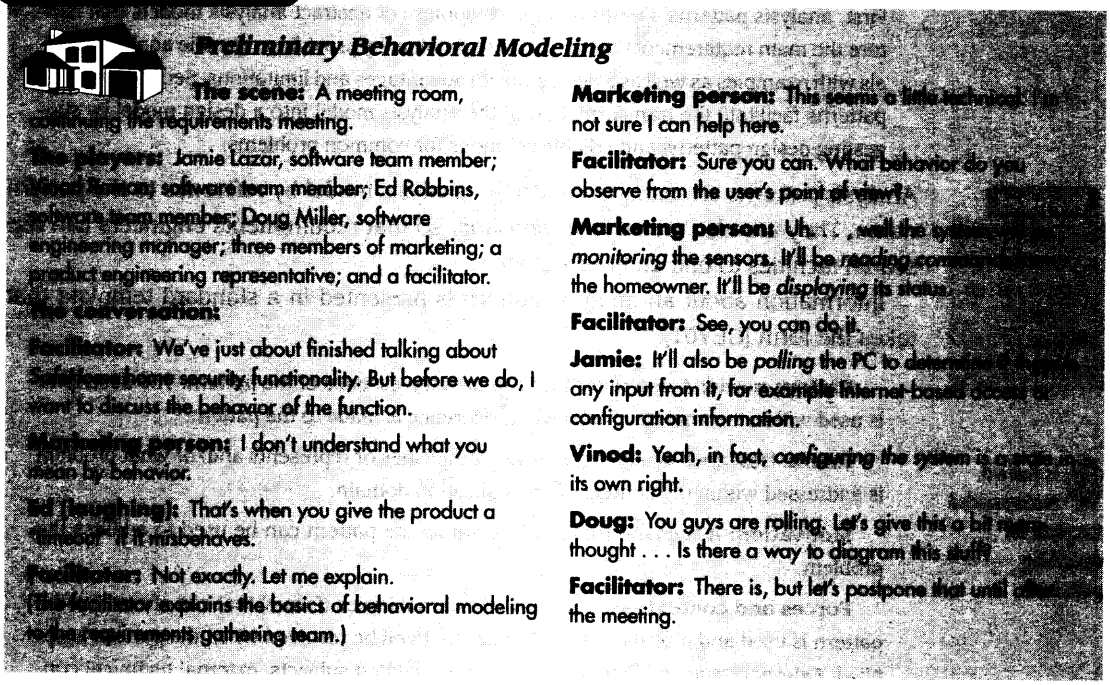
The *state diagram* (Chapter 8) is one method for representing the behavior of a system by depicting its states and the events that cause the system to change state. A *state* is any observable mode of behavior. In addition, the state diagram indicates what actions (e.g., process activation) are taken as a consequence of a particular event.

To illustrate a state diagram, consider a *reading commands* state for an office photocopier. UML state diagram notation is shown in Figure 7.6. A rounded rectangle represents a state. The rectangle is divided into three areas: (1) the state name (e.g., Reading commands), (2) *state variables* that indicate how the state manifests itself to the outside world, and (3) *state activities* that indicate how the state is entered (**entry/**) and actions (**do:**) that are invoked while in the state.

FIGURE 7.6
UML state diagram notation



SAFEHOME



Preliminary Behavioral Modeling

The scene: A meeting room, continuing the requirements meeting.

The players: Jamie Lazar, software team member; Vinod Raman, 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: We've just about finished talking about SafeHome home security functionality. But before we do, I want to discuss the behavior of the function.

Marketing person: I don't understand what you mean by behavior.

Ed (thoughtful): That's when you give the product a "stimulus" and it misbehaves.

Facilitator: Not exactly. Let me explain. (The facilitator explains the basics of behavioral modeling to the requirements gathering team.)

Marketing person: This seems a little technical. I'm not sure I can help here.

Facilitator: Sure you can. What behavior do you observe from the user's point of view?

Marketing person: Uh . . . well the system is *monitoring* the sensors. It'll be *reading* commands from the homeowner. It'll be *displaying* its status.

Facilitator: See, you can do it.

Jamie: It'll also be *polling* the PC to determine whether there's any input from it, for example Internet-based access or configuration information.

Vinod: Yeah, in fact, *configuring the system* is a big thing in its own right.

Doug: You guys are rolling. Let's give this a bit more thought . . . Is there a way to diagram this stuff?

Facilitator: There is, but let's postpone that until after the meeting.

Flow-oriented elements. Information is transformed as it flows through a computer-based system. The system accepts input in a variety of forms; applies functions to transform it; and produces output in a variety of forms. Input may be a control signal transmitted by a transducer, a series of numbers typed by a human operator, a packet of information transmitted on a network link, or a voluminous data file retrieved from secondary storage. The transform(s) may comprise a single logical comparison, a

complex numerical algorithm, or a rule-inference approach of an expert system. Output may light a single LED or produce a 200-page report. In effect, we can create a flow model for any computer-based system, regardless of size and complexity. A more detailed discussion of flow modeling is presented in Chapter 8.

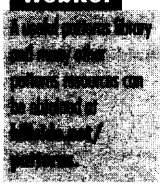
7.6.2 Analysis Patterns

Anyone who has done requirements engineering on more than a few software projects begins to notice that certain things reoccur across all projects within a specific application domain.[15] These can be called *analysis patterns* [FOW97] and represent something (e.g., a class, a function, a behavior) within the application domain that can be reused when modeling many applications.

Geyer-Schulz and Hahsler [GEY01] suggest two benefits that can be associated with the use of analysis patterns:

> First, analysis patterns speed up the development of abstract analysis models that capture the main requirements of the concrete problem by providing reusable analysis models with examples as well as a description of advantages and limitations. Second, analysis patterns facilitate the transformation of the analysis model into a design model by suggesting design patterns and reliable solutions for common problems.

WebRef



Analysis patterns are integrated into the analysis model by reference to the pattern name. They are also stored in a repository so that requirements engineers can use search facilities to find and reuse them.

Information about an analysis pattern is presented in a standard template that takes the form [GEY01]:[16]

Is there a recommended template for describing patterns?

Pattern name: A descriptor that captures the essence of the pattern. The descriptor is used within the analysis model when reference is made to the pattern.

Intent: Describes what the pattern accomplishes or represents and/or what problem is addressed within the context of an application domain.

Motivation: A scenario that illustrates how the pattern can be used to address the problem.

Forces and context: A description of external issues (forces) that can affect how the pattern is used and also the external issues that will be resolved when the pattern is applied. External issues can encompass business-related subjects, external technical constraints, and people-related matters.

Solution: A description of how the pattern is applied to solve the problem with an emphasis on structural and behavioral issues.

Consequences: Addresses what happens when the pattern is applied and what trade-offs exist during its application.

---

15 In some cases, things reoccur regardless of the application domain. For example, the features and functions of user interfaces are common regardless of the application domain under consideration.

16 A variety of patterns templates have been proposed in the literature. Interested readers should see [FOW97], [BUS96], and [GAM95] among many sources.