


**Design:** Discusses how the analysis pattern can be achieved through the use of known design patterns.

**Known uses:** Examples of uses within actual systems.

**Related patterns:** One or more analysis patterns that are related to the named pattern because the analysis pattern (1) is commonly used with the named pattern, (2) is structurally similar to the named pattern, (3) is a variation of the named pattern.

Examples of analysis patterns and further discussion of this topic are presented in Chapter 8.

INFO



### Patterns

We see patterns in virtually everything we encounter in everyday life.

Consider action-adventure movies—more specifically action-adventure detective movies with comic overtones. We can define patterns for *Hero&Sidekick*, *CaptainWhoManagesHero*, *CriminalwithaHeart*, and many more.

For example, *CaptainWhoManagesHero* is invariably older, wears a tie (Hero doesn't), yells at *Hero&Sidekick*

constantly, usually provides comic relief, or may be used in a more malevolent role to put bureaucratic or self-serving roadblocks in the way of *Hero&Sidekick*. A dramatic pattern is being established.

For a somewhat more technical example, consider a mobile phone. The following patterns are obvious: *MakeCall*, *LookUpNumber*, and *GetMessages* among many. Each of these patterns can be described once and then reused in software for any mobile phone.

In an ideal requirements engineering context, the inception, elicitation, and elaboration tasks determine customer requirements in sufficient detail to proceed to subsequent software engineering steps. Unfortunately, this rarely happens. In reality, the customer and the developer enter into a process of *negotiation*, where the customer may be asked to balance functionality, performance, and other product or system characteristics against cost and time to market. The intent of this negotiation is to develop a project plan that meets the needs of the customer while at the same time reflecting the real-world constraints (e.g., time, people, budget) that have been placed on the software team.

Negotiation is the art of dividing a cake in such a way that everyone believes he has the biggest piece.

The best negotiations strive for a “win-win” result.<sup>17</sup> That is, the customer wins by getting the system or product that satisfies the majority of the customer’s needs, and the software team wins by working to realistic and achievable budgets and deadlines.

<sup>17</sup> Dozens of books have been written on negotiating skills (e.g., [LEW00], [FAR97], [DON96]). It is one of the more important things that a young (or old) software engineer or manager can learn. Read one.

**WebRef**

A brief paper on negotiation for software requirements can be downloaded from [www.cse.cmu.edu/~negotiated/publications/Software\\_Requirements\\_Negotiation\\_Some\\_Insights\\_Learned.html](http://www.cse.cmu.edu/~negotiated/publications/Software_Requirements_Negotiation_Some_Insights_Learned.html)

Boehm [BOE98] defines a set of negotiation activities at the beginning of each software process iteration. Rather than a single customer communication activity, the following activities are defined:

1. Identification of the system or subsystem's key stakeholders.
2. Determination of the stakeholders' "win conditions."
3. Negotiate of the stakeholders' win conditions to reconcile them into a set of win-win conditions for all concerned (including the software team).

Successful completion of these initial steps achieves a win-win result, which becomes the key criterion for proceeding to subsequent software engineering activities.

**INFO****The Art of Negotiation**

Learning how to negotiate effectively can serve you well throughout your personal and technical life. The following guidelines are well worth considering:

1. *Recognize that it's not a competition.* To be successful, both parties have to feel they've won or achieved something. Both will have to compromise.
2. *Map out a strategy.* Decide what you'd like to achieve; what the other party wants to achieve, and how you'll go about making both happen.
3. *Listen actively.* Don't work on formulating your response while the other party is talking. Listen. It's likely you'll gain knowledge that will help you to better negotiate your position.
4. *Focus on the other party's interests.* Don't take hard positions if you want to avoid conflict.
5. *Don't let it get personal.* Focus on the problem that needs to be solved.
6. *Be creative.* Don't be afraid to think out of the box if you're at an impasse.
7. *Be ready to commit.* Once an agreement has been reached, don't waffle: commit to it and move on.

**SAFEHOME****The Start of a Negotiation**

**The scene:** Lisa Perez's office, after the first requirements gathering meeting.

**The players:** Doug Miller, software engineering manager, and Lisa Perez, marketing manager.

**Doug:** I hear the first meeting went really well.

**Lisa:** Actually, it did. You sent some good people to the meeting—they really contributed.

**Doug (laughing):** Yeah, they actually told me they got into it, and it wasn't a propeller head activity.

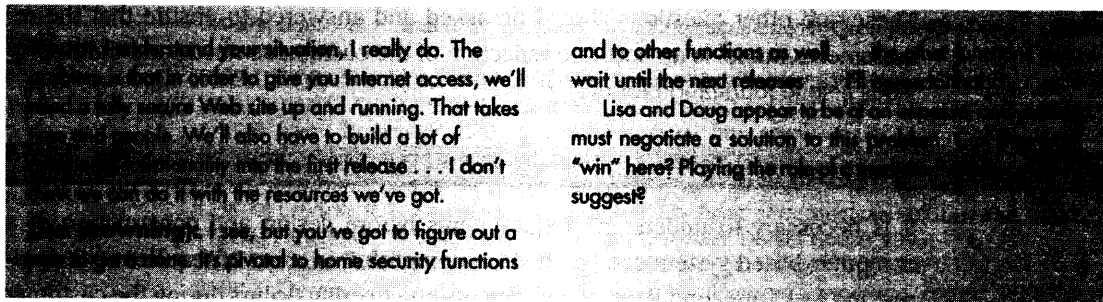
**Doug (laughing):** I'll be sure to take off my techie beans the next time I visit... Look, Lisa, I think we

may have a problem with getting all of the functionality for the home security function out by the dates your management is talking about. It's early, I know, but I've already been doing a little back of the envelope planning and...

**Lisa:** We've got to have it by that date, Doug. What functionality are you talking about?

**Doug:** I figure we can get full home security functionality out by the drop-dead date, but we'll have to delay Internet access till the second release.

**Lisa:** Doug, it's the Internet access that gives SafeHome "gee whizz" appeal. We're going to build our entire marketing campaign around it. We've gotta have it



## 7.6 VALIDATING REQUIREMENTS

As each element of the analysis model is created, it is examined for consistency, omissions, and ambiguity. The requirements represented by the model are prioritized by the customer and grouped within requirements packages that will be implemented as software increments and delivered to the customer. A review of the analysis model addresses the following questions:

**When I review requirements, what questions should I ask?**

- Is each requirement consistent with the overall objective for the system/product?
- Have all requirements been specified at the proper level of abstraction? That is, do some requirements provide a level of technical detail that is inappropriate at this stage?
- Is the requirement really necessary or does it represent an add-on feature that may not be essential to the objective of the system?
- Is each requirement bounded and unambiguous?
- Does each requirement have attribution? That is, is a source (generally, a specific individual) noted for each requirement?
- Do any requirements conflict with other requirements?
- Is each requirement achievable in the technical environment that will house the system or product?
- Is each requirement testable, once implemented?
- Does the requirements model properly reflect the information, function, and behavior of the system to be built?
- Has the requirements model been "partitioned" in a way that exposes progressively more detailed information about the system?
- Have requirements patterns been used to simplify the requirements model? Have all patterns been properly validated? Are all patterns consistent with customer requirements?

- 7.12.** What do you think happens when requirements validation uncovers an error? Who is involved in correcting the error?
- 7.13.** Using the template presented in Section 7.6.2, suggest one or more analysis patterns for an application suggested by your instructor.
- 7.14.** Describe an analysis pattern in your own words.
- 7.15.** What do use-case “exceptions” represent?
- 7.16.** What does “win-win” mean in the context of negotiation during the requirements engineering activity?
- 7.17.** Briefly discuss each of the elements of an analysis model. Indicate what each contributes to the model, how each is unique, and what general information is presented by each.

## FURTHER READINGS AND INFORMATION SOURCES

Because it is pivotal to the successful creation of any complex computer-based system, requirements engineering is discussed in a wide array of books. Hull and her colleagues (*Requirements Engineering*, Springer-Verlag, 2002), Bray (*An Introduction to Requirements Engineering*, Addison-Wesley, 2002), Arlow (*Requirements Engineering*, Addison-Wesley, 2001), Gilb (*Requirements Engineering*, Addison-Wesley, 2000), Graham (*Requirements Engineering and Rapid Development*, Addison-Wesley, 1999) and Sommerville and Kotonya (*Requirement Engineering: Processes and Techniques*, Wiley, 1998) are but a few of many books dedicated to the subject. Dan Berry (<http://se.uwaterloo.ca/~dberry/bib.html>) has published a wide variety of thought provoking papers on requirements engineering topics.

Lauesen (*Software Requirements: Styles and Techniques*, Addison-Wesley, 2002) presents a comprehensive survey of requirements analysis methods and notation. Weigers (*Software Requirements*, Microsoft Press, 1999) and Leffingwell and his colleagues (*Managing Software Requirements: A Unified Approach*, Addison-Wesley, 2000) present a useful collection of requirement best practices and suggest pragmatic guidelines for most aspects of the requirements engineering process.

Robertson and Robertson (*Mastering the Requirements Process*, Addison-Wesley, 1999) present a very detailed case study that helps to explain all aspects of the software requirements analysis and the analysis model. Kovitz (*Practical Software Requirements: A Manual of Content and Style*, Manning Publications, 1998) discusses a step-by-step approach to requirements analysis and a style guide for those who must develop requirements specifications. Jackson (*Software Requirements Analysis and Specification: A Lexicon of Practices, Principles and Prejudices*, Addison-Wesley, 1995) presents an intriguing look at the subject from A to Z (literally). Ploesch (*Assertions, Scenarios and Prototypes*, Springer-Verlag, 2003) discusses advanced techniques for developing software requirements.

Windle and Abreo (*Software Requirements Using the Unified Process*, Prentice-Hall, 2002) discuss requirements engineering within the context of the Unified Process and UML notation. Alexander and Steven (*Writing Better Requirements*, Addison-Wesley, 2002) present a brief set of guidelines for writing clear requirements, representing them as scenarios, and reviewing the end result.

Use-case modeling is often the driver for the creation of all other aspects of the analysis model. The subject is discussed at length by Bittner and Spence (*Use-Case Modeling*, Addison-Wesley, 2002), Cockburn [COC01], Armour and Miller (*Advanced Use-Case Modeling: Software Systems*, Addison-Wesley, 2000), Kulak and his colleagues (*Use Cases: Requirements in Context*, Addison-Wesley, 2000), and Schneider and Winters (*Applying Use Cases*, Addison-Wesley, 1998).

A wide variety of information sources on requirements engineering and analysis are available on the Internet. An up-to-date list of World Wide Web references that are relevant to requirements engineering and analysis can be found at the SEPA Web site:

**<http://www.mhhe.com/pressman>**.

# BUILDING THE ANALYSIS MODEL

## KEY CONCEPTS

analysis modeling

behavioral

class-based

control spec

CRC

data

flow-oriented

scenario-based

classes

data flow diagram

data objects

domain analysis

structured analysis

rules of thumb

**A**t a technical level, software engineering begins with a series of modeling tasks that lead to a specification of requirements and a comprehensive design representation for the software to be built. The *analysis model*, actually a set of models, is the first technical representation of a system.

In a seminal book on analysis modeling methods, Tom DeMarco [DEM79] describes the process in this way:

Looking back over the recognized problems and failings of the analysis phase, I suggest that we need to make the following additions to our set of analysis phase goals:

- The products of analysis must be highly maintainable. This applies particularly to the Target Document [software requirements specification].
- Problems of size must be dealt with using an effective method of partitioning. The Victorian novel specification is out.
- Graphics have to be used whenever possible.
- We have to differentiate between logical [essential] and physical [implementation] considerations . . . .

At the very least, we need . . .

- Something to help us partition our requirements and document that partitioning before specification . . .
- Some means of keeping track of and evaluating interfaces . . .
- New tools to describe logic and policy, something better than narrative text.

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

## 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. Analysis modeling uses a combination of text and diagrammatic forms to depict requirements for data, function, and behavior 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. Analysis modeling represents requirements in multiple dimensions, thereby increasing the probability that errors will be found, that inconsistency will surface, and that omissions will be uncovered.

### 8.1.2 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:

- *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 analysis 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 non-functional models until design. For example, 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 "interconnect- edness" is extremely high, efforts should be made to reduce it.*
- *Be certain that the analysis model provides value to all stakeholders. Each constituency has its own use for the model. For example, business stake- holders 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 provide no new information. Don't use complete notational forms, when a simple list will do.*

### 8.1.3 Domain Analysis

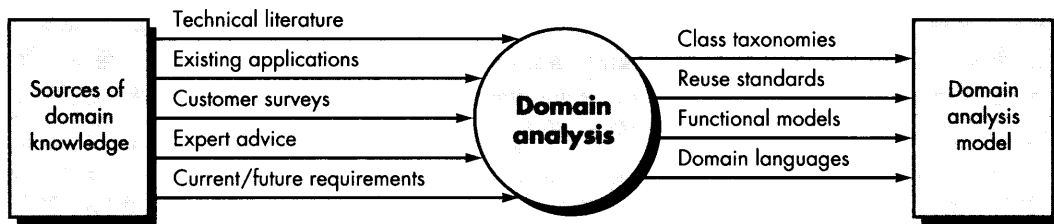
In our discussion of requirements engineering (Chapter 7), we noted that analysis patterns often reoccur across many applications within a specific business domain. If these patterns are defined and categorized in a manner that allows a software engineer or analyst to recognize and reuse them, the creation of the analysis model is expedited. More important, the likelihood of applying reusable design patterns and executable software components grows dramatically. This improves time to market and reduces development costs.

But how are analysis patterns recognized in the first place? Who defines them, categorizes them, and readies them for use on subsequent projects? The answers to these questions lies in *domain analysis*. Firesmith [FIR93] describes domain analysis in the following way:

Software domain analysis is the identification, analysis, and specification of common re- quirements from a specific application domain, typically for reuse on multiple projects

#### WebRef

Many useful resources for domain analysis can be found at [www.itwrls.com/English/SoftwareEngineering/SE\\_mod5.asp](http://www.itwrls.com/English/SoftwareEngineering/SE_mod5.asp).

**FIGURE 8.2** Input and output for domain analysis

within that application domain . . . [Object-oriented domain analysis is] the identification, analysis, and specification of common, reusable capabilities within a specific application domain, in terms of common objects, classes, subassemblies, and frameworks.

The “specific application domain” can range from avionics to banking, from multimedia video games to software embedded within medical devices. The goal of domain analysis is straightforward: to find or create those analysis classes and/or common functions and features that are broadly applicable, so that they may be reused.<sup>3</sup>

“The great art of learning is to understand but little at a time.”

John Locke

#### WebRef

A worthwhile discussion of domain engineering and analysis can be found at [www.scl.com.edu/stu/descriptions/deda.html](http://www.scl.com.edu/stu/descriptions/deda.html).

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<sup>4</sup> is to discover and define reusable analysis patterns, analysis classes, and related information that may be used by many people working on similar but not necessarily the same applications.

Figure 8.2 [ARA89] illustrates key inputs and outputs for the domain analysis process. Sources of domain knowledge are surveyed in an attempt to identify objects that can be reused across the domain.

## 8.2 ANALYSIS MODELING APPROACHES

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

3 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” [LET03].

4 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.

### KEY POINT

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

### WebRef

A concept called "normalization" is important to those who intend to do thorough data modeling. A useful introduction can be found at [www.datamodel.org](http://www.datamodel.org).

### 8.3.2 Data Attributes

*Data attributes* define the properties of a data object and take on one of three different characteristics. They can be used to (1) name an instance of the data object, (2) describe the instance, or (3) make reference to another instance in another table. In addition, one or more of the attributes must be defined as an identifier—that is, the identifier attribute becomes a "key" when we want to find an instance of the data object. In some cases, values for the identifier(s) are unique, although this is not a requirement. Referring to the data object **car**, a reasonable identifier might be the ID number.

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



### Data Objects and OO Classes— Are They the Same Thing?

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

A data object defines a composite data item; that is, it incorporates a collection of individual data items (attributes) and gives the collection of items a name (the name of the data object). An OO class encapsulates data attributes but also incorporates the operations that

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

### INFO

### KEY POINT

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

### 8.3.3 Relationships

Data objects are connected to one another in different ways. Consider the two data objects, **person** and **car**. These objects can be represented using the simple notation illustrated in Figure 8.5a. A connection is established between **person** and **car** because the two objects are related. But what are the relationships? To determine the answer, we must understand the role of people (owners, in this case) and cars within the context of the software to be built. We can define a set of object/relationship pairs that define the relevant relationships. For example,

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

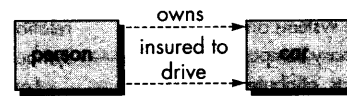


FIGURE 8.5

Relationships  
between data  
objects



(a) A basic connection between data objects




(b) Relationships between data objects

The relationships *owns* and *insured to drive* define the relevant connections between **person** and **car**. Figure 8.5b illustrates these object/relationship pairs graphically. The arrows noted in Figure 8.5b provide important information about the directionality of the relationship and often reduce ambiguity or misinterpretations.

### 8.3.4 Cardinality and Modality

The elements of data modeling—data objects, attributes, and relationships—provide the basis for understanding the information domain of a problem. However, additional information related to these basic elements must also be understood.

We have defined a set of objects and represented the object/relationship pairs that bind them. But a simple pair that states that **objectX** *relates to* **objectY** does not provide enough information for software engineering purposes. We must understand how many occurrences of **objectX** are related to how many occurrences of **objectY**. This leads to a data modeling concept called *cardinality*.

 How do I handle a situation in which one data object is related to many occurrences of another data object?

The data model must be capable of representing the number of occurrences of objects in a given relationship. Tillmann [TIL93] defines the cardinality of an object/relationship pair in the following manner: “Cardinality is the specification of the number of occurrences of one [object] that can be related to the number of occurrences of another [object].” For example, one object can relate to only one other object (a 1:1 relationship); one object can relate to many objects (a 1:N relationship); some number of occurrences of an object can relate to some other number of occurrences of another object (an M:N relationship).<sup>6</sup> Cardinality also defines “the maximum number of objects that can participate in a relationship” [TIL93]. It does not, however, provide an indication of whether or not a particular data object must participate in the relationship. To specify this information, the data model adds modality to the object/relationship pair.

<sup>6</sup> For example, an uncle can have many nephews, and a nephew can have many uncles.

**Class**—encapsulates the data and procedural abstractions required to describe the content and behavior of some real world entity. Stated another way, a class is a generalized description (e.g., a template, pattern, or blueprint) that describes a collection of similar objects.

**Objects**—instances of a specific class. Objects inherit a class' attributes and operations.

**Operations**—also called *methods* and *services*, provide a representation of one of the behaviors of a class.

**Subclass**—a specialization of the superclass. A subclass can inherit both attributes and operations from a superclass.

**Superclass**—also called a *base class*, is a generalization of a set of classes that are related to it.

## 8.5 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 software engineers understand how end-users (and other actors) want to interact with a system, the software team will be better able to properly characterize requirements and build meaningful analysis and design models. Hence, analysis modeling with UML begins with the creation of scenarios in the form of use-cases, activity diagrams, and swimlane diagrams.

### 8.5.1 Writing Use-Cases

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.<sup>9</sup>

The concept of a use-case (Chapter 7) is relatively easy to understand—describe a specific usage scenario in straightforward language from the point of view of a defined actor.<sup>10</sup> But how do we know (1) what to write about, (2) how much to write about it, (3) how detailed to make our description, and (4) how to organize the description? These are the questions that must be answered if use-cases are to provide value as an analysis modeling tool.



*In some situations, use-cases become the dominant requirements engineering mechanism. However, this does not mean that you should discard the concepts and techniques discussed in Chapter 7.*

**"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**

**What to write about?** The first two requirements engineering tasks<sup>11</sup>—inception and elicitation—provide us the information we need to begin writing use cases. Requirements gathering meetings, QFD, and other requirements engineering mecha-

<sup>9</sup> Use-cases are a particularly important part of analysis modeling for user interfaces. Interface analysis is discussed in detail in Chapter 12.

<sup>10</sup> 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" [COC01].

<sup>11</sup> These requirements engineering tasks are discussed in detail in Chapter 7.

nisms are used to identify stakeholders, define the scope of the problem, specify overall operational goals, 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, the functions or activities performed by a specific actor are listed. These may be obtained from a list of required system functions, through conversations with customers or end-users, or by an evaluation of activity diagrams (Section 8.5.2) developed as part of analysis modeling.

## SAFEHOME



### Developing Another Preliminary User Scenario

Ed is in a meeting room, during the second preliminary gathering meeting:

**Ed:** I brought James Laxar, software team member; Ed Roberts, 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 home surveillance function. Let's develop a user scenario for access to the home security function.

**Ed:** 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 of the actor?

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

**Facilitator:** Right . . . same way.

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

**Ed:** Will the video be digital, and will it be stored on disk?

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

**Meredith:** Okay, so basically there are two parts to the surveillance function . . . the first configures the system

including laying out a floor plan, we need tools to help the homeowner do this—and the second part is the actual surveillance function itself. So for the first part of the configuration activity, I'll have an actor who plays the role of the actor.

**Facilitator (smiling):** Took the words right out of your 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 request camera output and replay camera output. I also want the ability to block access to one or more cameras with a specific password. And I 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 from all the cameras. I also want the interface to 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 the function 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:

- Access camera surveillance via the Internet.

camera” results in an error condition: “No floor plan configured for this house.”<sup>12</sup> This error condition becomes a secondary scenario.

*Is it possible that the actor will encounter some other behavior at this point?* Again the answer to the question is yes. As steps 6 and 7 occur, the system may encounter an alarm condition. This would result in the system displaying a special alarm notification (type, location, system action) and providing the actor with a number of options relevant to the nature of the alarm. Because this secondary scenario can occur 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.

Referring to the formal use-case template shown in the sidebar, the secondary scenarios are represented as exceptions to the basic sequence described for **ACS-DCV**.

SAFEHOME

### Use-Case Template for Surveillance

**Use Case:** Access camera surveillance—display camera views (ACS-DCV).

**Primary actor:** Homeowner.

**Goal or 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.

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

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 “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 cameras”—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.

<sup>12</sup> 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 to be certain that each is accessible via the system and through the floor plan.

Channel to actor:	Via PC-based browser and Internet connection to <i>SafeHome</i> Web site.	2. Is security sufficient? Hacking into this system represent a major invasion of privacy.
Secondary factors:	System administrator, cameras.	3. Will system response via the Internet be acceptable given the bandwidth required for camera views?
Channels to secondary actors:		4. Will we develop a capability to provide higher frames-per-second rate when high bandwidth connections are available?
1. System administrator: PC-based system		
2. Cameras: wireless connectivity		
Open issues:		
1. What mechanisms protect unauthorized use of this capability by employees of the company?		

**WebRef**

When are you finished writing use-cases? For a worthwhile discussion of this topic, see [eotips.org/use-cases-done.html](http://eotips.org/use-cases-done.html).

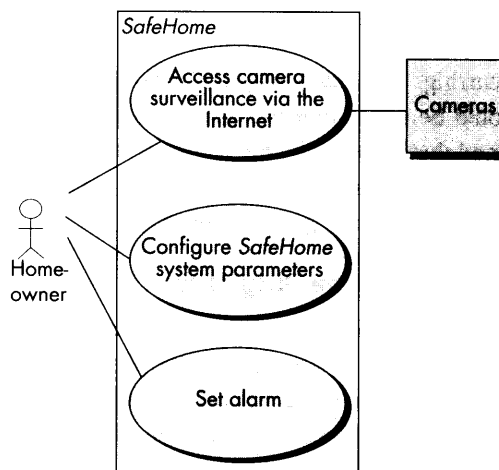
In many cases, there is no need to create a graphical representation of a usage scenario. However, diagrammatic representation can facilitate understanding, particularly when the scenario is complex. As we noted in Chapter 7, UML does provide use-case diagramming capability. Figure 8.6 depicts a preliminary use-case diagram for the *SafeHome* product. Each use-case is represented by an oval. Only the use-case, **ACS-DCV** has been discussed in detail in this section.

**8.5.2 Developing an Activity Diagram**

The UML activity diagram (discussed briefly in Chapters 6 and 7) 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

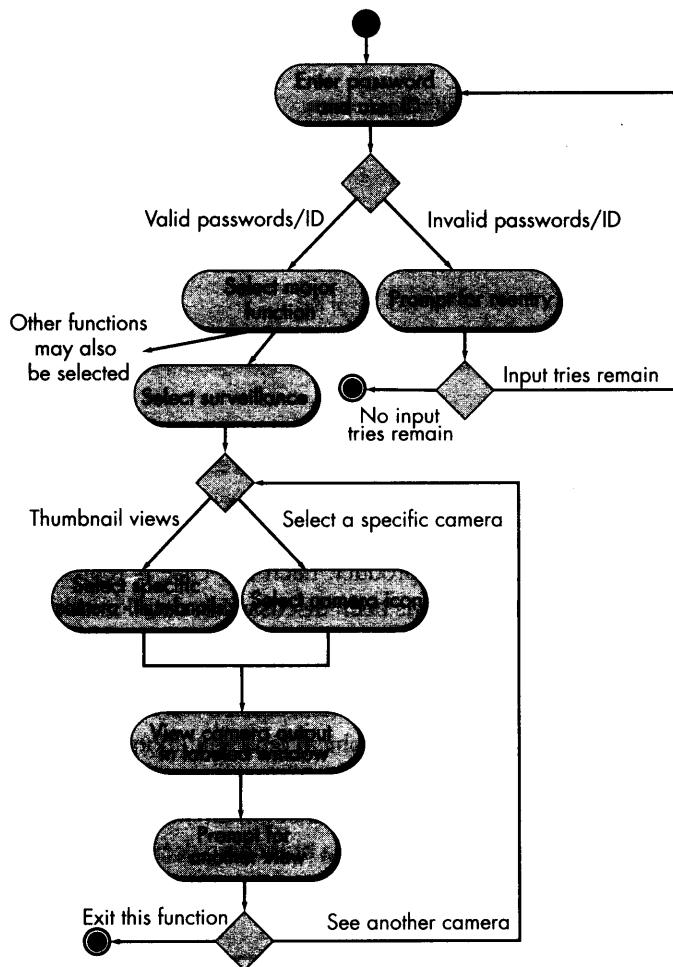
**FIGURE 8.6**

Preliminary use-case diagram for the *SafeHome* system



**FIGURE 8.7**

**Activity diagram for Access camera surveillance—display camera views function**



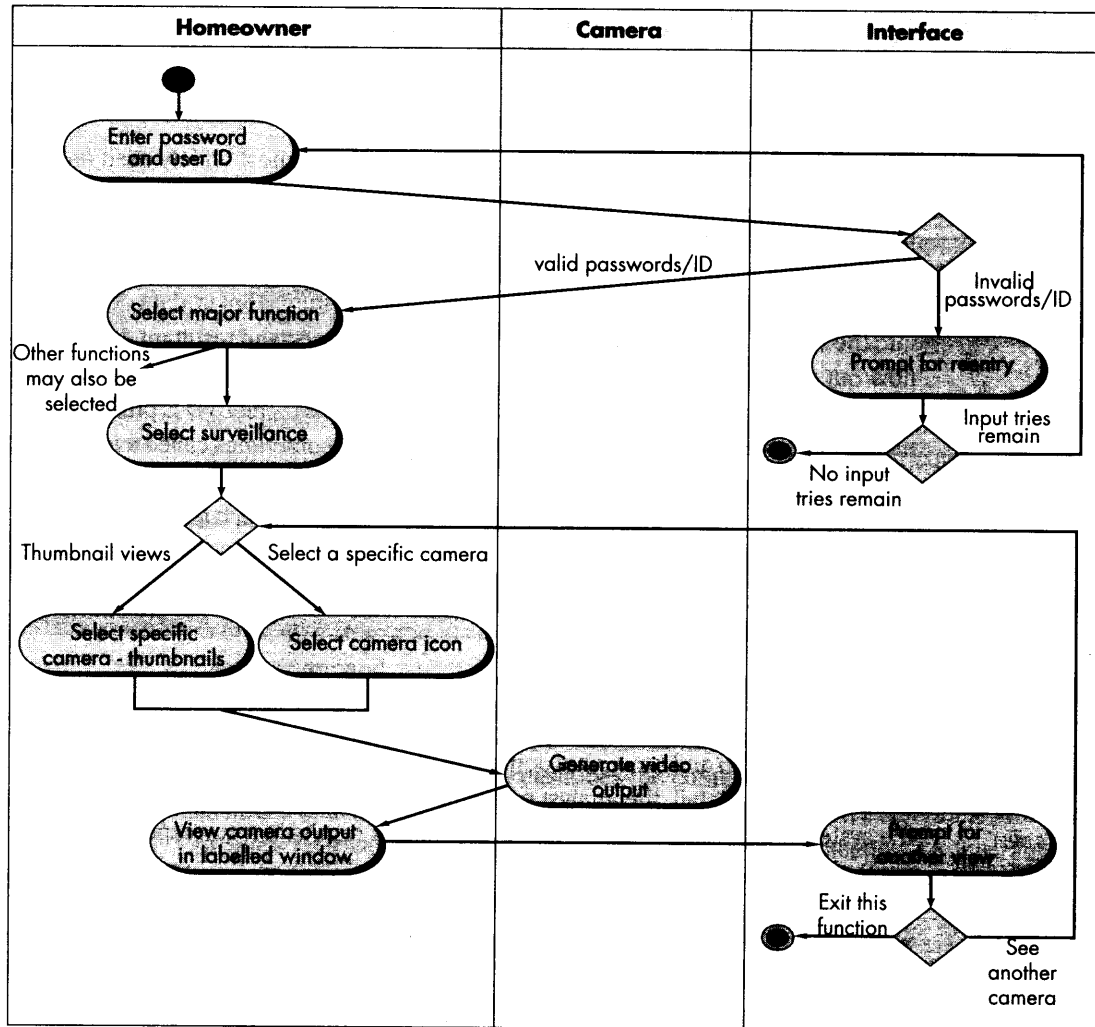
**KEY POINT**

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

occurring. An activity diagram for the **ACS-DCV** function is shown in Figure 8.7. 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*.

**8.5.3 Swimlane Diagrams**

The UML *swimlane diagram* is a useful variation of the activity diagram and allows the modeler 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 function) or analysis class has responsibility for the action described by an activity

**FIGURE 8.8** Swimlane diagram for Access camera surveillance—display camera views function**KEY POINT**

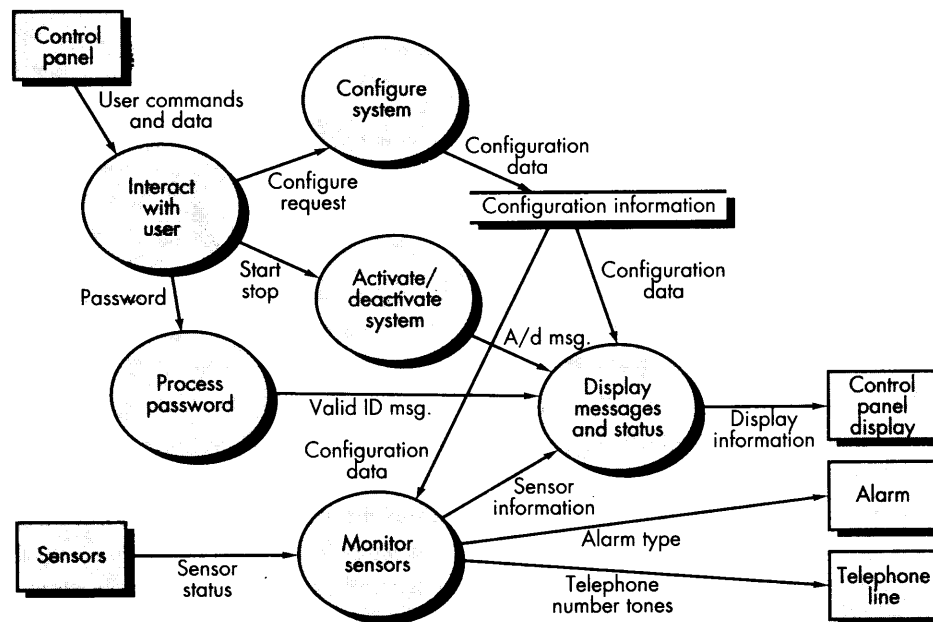
A UML swimlane diagram represents the flow of actions and decisions and indicates which actors perform each.

rectangle. Responsibilities are represented as parallel segments that divide the diagram vertically, like the lanes in a swimming pool.

Three analysis classes—**Homeowner**, **Interface**, and **Camera**—have direct or indirect responsibilities in the context of the activity diagram represented in Figure 8.7. Referring to Figure 8.8, the activity diagram is rearranged so that activities associated with a particular analysis class fall inside the swimlane for that class. For example, the **Interface** class represents the user interface as seen by the homeowner. The activity diagram notes two prompts that are the responsibility of the interface—*prompt for*

FIGURE 8.10

Level 1 DFD for SafeHome security function



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

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

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



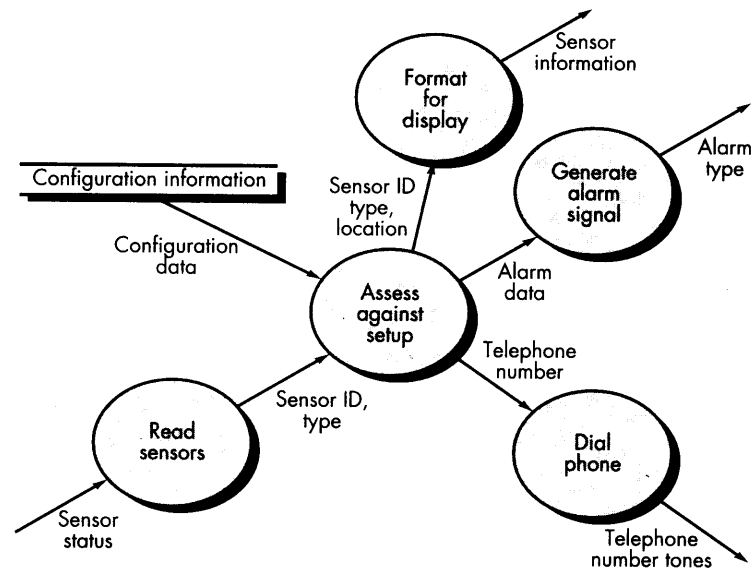
Be certain that the processing narrative you intend to parse is written at the same level of abstraction throughout.

Referring to the grammatical parse, a pattern begins to emerge. Verbs are *SafeHome* processes; that is, they may ultimately be represented as bubbles in a subsequent DFD. Nouns are either external **entities** (boxes), data or control objects (arrows), or data stores (double lines). Note further that nouns and verbs can be associated with one another. For example, each sensor is assigned a number and type, therefore **number** and **type** are attributes of the data object **sensor**. Therefore, by performing a grammatical parse on the processing narrative for a bubble at any DFD level, we can generate much useful information about how to proceed with the refinement to the next level. Using this information, a level 1 DFD is shown in Figure 8.10. The context level process shown in Figure 8.9 has been expanded into six processes derived from an ex-



FIGURE 8.11

Level 2 DFD that refines the monitor sensors process



amination of the grammatical parse. Similarly, the information flow between processes at level 1 has been derived from the parse. In addition, information flow continuity is maintained between levels 0 and 1.

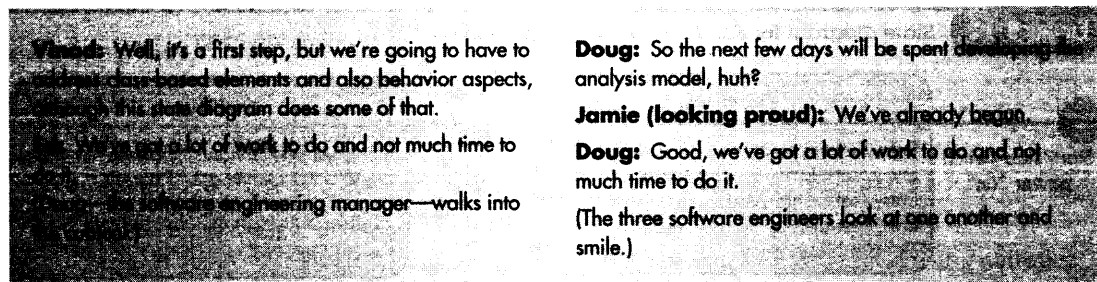
The processes represented at DFD level 1 can be further refined into lower levels. For example, the process *monitor sensors* can be refined into a level 2 DFD as shown in Figure 8.11. Note once again that information flow continuity has been maintained between levels.

The refinement of DFDs continues until each bubble performs a simple function. That is, until the process represented by the bubble performs a function that would be easily implemented as a program component. In Chapter 9, we discuss a concept, called *cohesion*, that can be used to assess the simplicity of a given function. For now, we strive to refine DFDs until each bubble is “single-minded.”

### 8.6.2 Creating a Control Flow Model

For many types of applications, the data model and the data flow diagram are all that is necessary to obtain meaningful insight into software requirements. As we have already noted, however, a large class of applications are “driven” by events rather than data, produce control information rather than reports or displays, and process information with heavy concern for time and performance. Such applications require the use of *control flow modeling* in addition to data flow modeling.

We have already noted that an event or control item is implemented as a Boolean value (e.g., true or false, on or off, 1 or 0) or a discrete list of conditions



#### 8.6.4 The Process Specification

##### KEY POINT

The PSPEC is a “mini-specification” for each transform at the lowest refined level of a DFD.

The *process specification* (PSPEC) is used to describe all flow model processes that appear at the final level of refinement. The content of the process specification can include narrative text, a program design language (PDL) description<sup>19</sup> of the process algorithm, mathematical equations, tables, diagrams, or charts. By providing a PSPEC to accompany each bubble in the flow model, the software engineer creates a “mini-spec” that can serve as a guide for design of the software component that will implement the process.

To illustrate the use of the PSPEC, consider the *process password* transform represented in the flow model for *SafeHome* (Figure 8.10). The PSPEC for this function might take the form:

**PSPEC: process password (at control panel).** The *process password* transform performs password validation at the control panel for the *SafeHome* security function. *Process password* receives a four-digit password from the *interact with user* function. The password is first compared to the master password stored within the system. If the master password matches, [valid id message = true] is passed to the *message and status display* function. If the master password does not match, the four digits are compared to a table of secondary passwords (these may be assigned to house guests and/or workers who require entry to the home when the owner is not present). If the password matches an entry within the table, [valid id message = true] is passed to the *message and status display* function. If there is no match, [valid id message = false] is passed to the message and status display function.

If additional algorithmic detail is desired at this stage, a program design language representation may also be included as part of the PSPEC. However, many believe that the PDL version should be postponed until component design commences.

<sup>19</sup> Program design language (PDL) mixes programming language syntax with narrative text to provide procedural design detail. PDL is discussed in Chapter 11.

## SOFTWARE TOOLS

**Structured Analysis**

**Objective:** Structured analysis tools allow a software engineer to create data models, flow models, and behavioral models in a manner that enables consistency and continuity checking and easy editing and extension. Models created using these tools provide the software engineer with insight into the analysis representation and help to eliminate errors before they propagate into design, or worse, into implementation itself.

**Mechanics:** Tools in this category use a “data dictionary” as the central database for the description of all data objects. Once entries in the dictionary are defined, entity-relationship diagrams can be created and object hierarchies can be developed. Data flow diagramming features allow easy creation of this graphical model and also provide features for the creation of PSPECs and CSPECs. Analysis tools also enable the software engineer to create behavioral models using the state diagram as the operative notation.

**Representative Tools<sup>20</sup>**

*AxiomSys*, developed by STG, Inc. ([www.stgcase.com](http://www.stgcase.com)), provides a complete structure analysis tools suite including Hatley-Pirbhai extensions for the modeling of real-time systems.

*MacA&D*, *WinA&D* developed by Excel Software ([www.excelsoftware.com](http://www.excelsoftware.com)), provides a set of simple and inexpensive analysis and design tools for Macs and Windows machines.

*MetaCASE Workbench*, developed by MetaCase Consulting ([www.metacase.com](http://www.metacase.com)), is a metatool used to define an analysis or design method (including structured analysis): its concepts, rules, notations, and generators.

*System Architect*, developed by Popkin Software ([www.popkin.com](http://www.popkin.com)), provides a broad range of analysis and design tools including tools for data modeling and structured analysis.

**8.7 CLASS-BASED MODELING**

How do we go about developing the class-based elements of an analysis model—classes and objects, attributes, operations, packages, CRC models, and collaboration diagrams? The sections that follow present a series of informal guidelines that will assist in their identification and representation.

**8.7.1 Identifying Analysis Classes**

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

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

*Carl Ayala*

We can begin to identify classes by examining the problem statement or (using the terminology applied earlier in this chapter) by performing a “grammatical parse” on

<sup>20</sup> 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.

the use-cases or processing narratives developed for the system to be built. Classes are determined by underlining each noun or noun clause and entering it into a simple table. Synonyms should be noted. If the class is required to implement a solution, then it is part of the solution space; otherwise, if a class is necessary only to describe a solution, it is part of the problem space. What should we look for once all of the nouns have been isolated? *Analysis classes* manifest themselves in one of the following ways:

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

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

This categorization is but one of many that have been proposed in the literature.<sup>21</sup> For example, Budd [BUD96] suggests a taxonomy of classes that includes producers (sources) and consumers (sinks) of data, data managers, view or observer classes, and helper classes.

It is also important to note what classes or objects are not. In general, a class should never have an “imperative procedural name” [CAS89]. For example, if the developers of software for a medical imaging system defined an object with the name **InvertImage** or even **ImageInversion**, they would be making a subtle mistake. The **Image** obtained from the software could, of course, be a class (it is a thing that is part of the information domain). Inversion of the image is an operation that is applied to the class. It is likely that *inversion()* would be defined as an operation for the class **Image**, but it would not be defined as a separate class to connote “image inversion.” As Cashman [CAS89] states: “the intent of object-orientation is to encapsulate, but still keep separate, data and operations on the data.”

To illustrate how analysis classes might be defined during the early stages of modeling, we return to the *SafeHome* security function. In Section 8.6.1, we performed a

<sup>21</sup> Another important categorization—defining entity, boundary, and controller classes—is discussed in Section 8.7.4.

“grammatical parse” on a processing narrative<sup>22</sup> for the security function. Extracting the nouns, we can propose a number of potential classes:

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

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

*“Classes struggle, some classes triumph, others are eliminated.”*

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

Coad and Yourdon [COA91] suggest six selection characteristics that should be used as an analyst considers each potential class for inclusion in the analysis model:

1. *Retained information.* The potential class will be useful during analysis only if information about it must be remembered so that the system can function.
2. *Needed services.* The potential class must have a set of identifiable operations that can change the value of its attributes in some way.
3. *Multiple attributes.* During requirement analysis, the focus should be on “major” information; a class with a single attribute may, in fact, be useful during design, but is probably better represented as an attribute of another class during the analysis activity.
4. *Common attributes.* A set of attributes can be defined for the potential class, and these attributes apply to all instances of the class.

<sup>22</sup> It is important to note that this technique should also be used for every use-case developed as part of the requirements gathering (elicitation) activity. That is, use-cases can be grammatically parsed to extract potential analysis classes.

5. *Common operations.* A set of operations can be defined for the potential class, and these operations apply to all instances of the class.
6. *Essential requirements.* External entities that appear in the problem space and produce or consume information essential to the operation of any solution for the system will almost always be defined as classes in the requirements model.

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

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

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

### KEY POINT

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

#### 8.7.2 Specifying Attributes

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

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

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

```

identification information = system ID + verification phone number + system status
alarm response information = delay time + telephone number
activation/deactivation information = master password + number of allowable tries +
temporary password

```

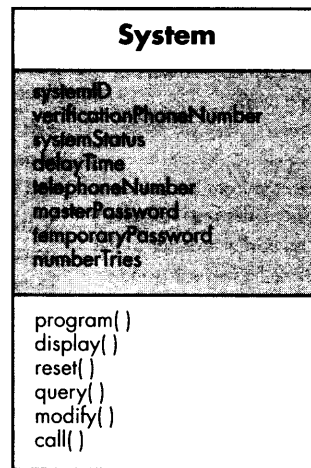
Some of the data items to the right of the equal sign could be further refined to an elementary level, but for our purposes, they constitute a reasonable list of attributes for the **System** class (shaded portion of Figure 8.13).

Sensors are part of the overall *SafeHome* system, and yet they are not listed as data items or as attributes in Figure 8.13. **Sensor** has already been defined as a class, and multiple **Sensor** objects will be associated with the **System** class. In general, we avoid defining an item as an attribute if more than one of the items is to be associated with the class.

### 8.7.3 Defining Operations

Operations define the behavior of an object. Although many different types of operations exist, they can generally be divided into three broad categories: (1) operations

**FIGURE 8.13**  
Class diagram  
for the system  
class





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

that manipulate data in some way (e.g., adding, deleting, reformatting, selecting), (2) operations that perform a computation, (3) operations that inquire about the state of an object, and (4) operations that monitor an object for the occurrence of a controlling event. These functions are accomplished by operating on attributes and/or associations (Section 8.7.5). Therefore, an operation must have “knowledge” of the nature of the class’ attributes and associations.

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

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

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

## SAFEHOME



### Class Models

The *keenest*. Ed’s cubicle, as analysis

Ed, Vinod, and Ed—all members of the engineering team.

Ed is looking to extract classes from the use-case “camera surveillance—display” presented in an earlier sidebar in this chapter. Here’s the diagram.

Ed: The homeowner wants to pick a camera, he wants to pick a sensor from a floor plan. I’ve defined a class for each. Here’s the diagram.

(Figure 8.14.1)

Jamie: So **FloorPlan** is a class that is composed of walls that are composed of wall segments, doors, windows, and also cameras; that’s what those lines mean, right?

Ed: Yeah, they’re called “associations.” One floor is associated with another according to the associations I’ve shown. [Associations are discussed in Section 8.7.5.]

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

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



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

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

**Vinod:** I have a question. Why does the camera have an ID but the others don't?

**Ed:** We'll need to identify each camera for display purposes.

**Jamie:** Makes sense to me, but I do have a few more questions.

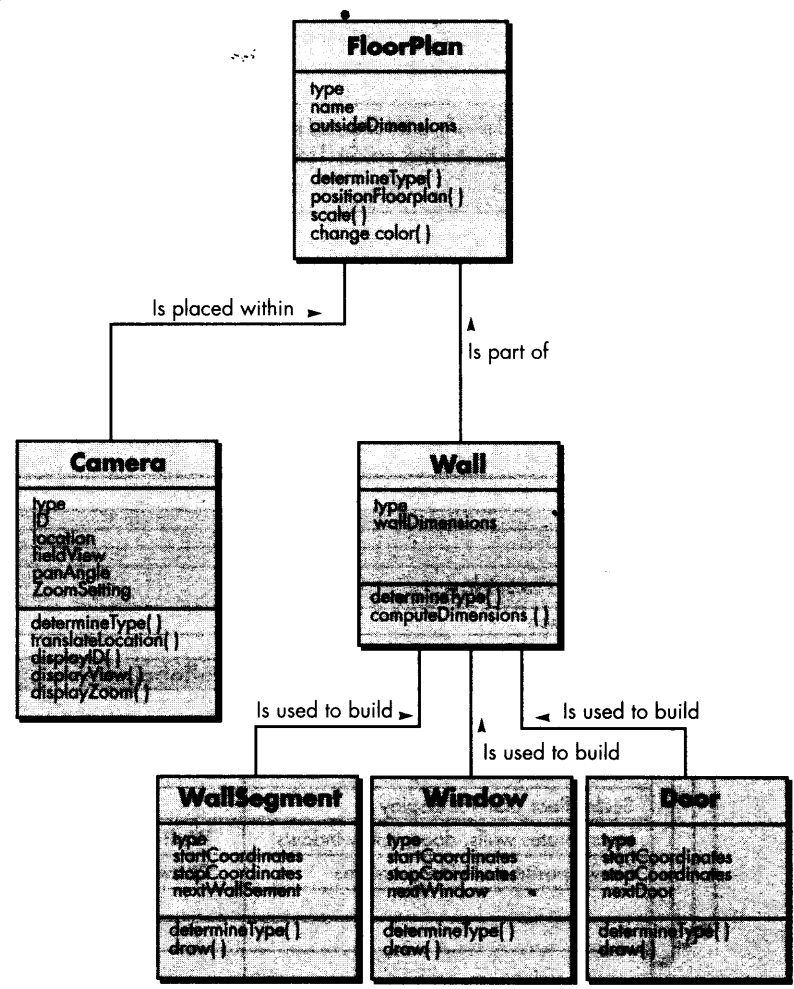
(Jamie asks questions which result in minor modifications.)

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

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

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

**FIGURE 8.14** Class diagram for FloorPlan (see sidebar discussion)



### 8.7.4 Class-Responsibility-Collaborator (CRC) Modeling

*Class-responsibility-collaborator (CRC) modeling* [WIR90] provides a simple means for identifying and organizing the classes that are relevant to system or product requirements. Ambler [AMB95] describes CRC modeling in the following way:

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

In reality, the CRC model may make use of actual or virtual index cards. The intent is to develop an organized representation of classes. *Responsibilities* are the attributes and operations that are relevant for the class. Stated simply, a responsibility is “anything the class knows or does” [AMB95]. *Collaborators* are those classes that are required to provide a class with the information needed to complete a responsibility. In general, a collaboration implies either a request for information or a request for some action.

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

**C. Horstmann**

A simple CRC index card for the **FloorPlan** class is illustrated in Figure 8.15. The list of responsibilities shown on the CRC card is preliminary and subject to additions or modification. The classes **Wall** and **Camera** are noted next to the responsibility that will require their collaboration.

**FIGURE 8.15**

A CRC model index card

Class: FloorPlan	
Description	
Responsibility:	Collaborator:
Defines floor plan name/type	
Manages floor plan positioning	
Scales floor plan for display	
Scales floor plan for display	
Incorporates walls, doors and windows	<b>Wall</b>
Shows position of video cameras	<b>Camera</b>

**Classes.** Basic guidelines for identifying classes and objects have been presented earlier in this chapter. The taxonomy of class types presented in Section 8.7.1 can be extended by considering the following categories:

#### WebRef

An excellent discussion of these class types can be found at [www.thecomcafe.com/a0079.htm](http://www.thecomcafe.com/a0079.htm).

- *Entity classes*, also called *model* or *business* classes, are extracted directly from the statement of the problem (e.g., **FloorPlan** and **Sensor**). These classes typically represent things that are to be stored in a database and persist throughout the duration of the application (unless they are specifically deleted).
- *Boundary classes* are used to create the interface (e.g., interactive screen or printed reports) that the user sees and interacts with as the software is used. Entity classes contain information that is important to users, but they do not display themselves. Boundary classes are designed with the responsibility of managing the way entity objects are represented to users. For example, a boundary class called **CameraWindow** would have the responsibility of displaying surveillance camera output for the *SafeHome* system.
- *Controller classes* manage a “unit of work” [UML03] from start to finish. That is, controller classes can be designed to manage (1) the creation or update of entity objects; (2) the instantiation of boundary objects as they obtain information from entity objects; (3) complex communication between sets of objects; and (4) validation of data communicated between objects or between the user and the application. In general, controller classes are not considered until design has begun.

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

Russell Baker

**Responsibilities.** Basic guidelines for identifying responsibilities (attributes and operations) have been presented in Sections 8.7.2 and 8.7.3. Wirfs-Brock and her colleagues [WIR90] suggest five guidelines for allocating responsibilities to classes:

 What guidelines can be applied for allocating responsibilities to classes?

1. **System intelligence should be distributed across classes to best address the needs of the problem.** Every application encompasses a certain degree of intelligence, that is, what the system knows and what it can do. This intelligence can be distributed across classes in a number of different ways. “Dumb” classes (those that have few responsibilities) can be modeled to act as servants to a few “smart” classes (those having many responsibilities). Although this approach makes the flow of control in a system straightforward, it has a few disadvantages: (a) it concentrates all intelligence within a few classes, making changes more difficult, and (b) it tends to require more classes, hence more development effort.

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

To determine whether system intelligence is properly distributed, the responsibilities noted on each CRC model index card should be evaluated to determine if any class has an extraordinarily long list of responsibilities. This indicates a concentration of intelligence.<sup>23</sup> In addition, the responsibilities for each class should exhibit the same level of abstraction.

2. **Each responsibility should be stated as generally as possible.** This guideline implies that general responsibilities (both attributes and operations) should reside high in the class hierarchy (because they are generic, they will apply to all subclasses).
3. **Information and the behavior related to it should reside within the same class.** This achieves the OO principle called *encapsulation*. Data and the processes that manipulate the data should be packaged as a cohesive unit.
4. **Information about one thing should be localized with a single class, not distributed across multiple classes.** A single class should take on the responsibility for storing and manipulating a specific type of information. This responsibility should not, in general, be shared across a number of classes. If information is distributed, software becomes more difficult to maintain and more challenging to test.
5. **Responsibilities should be shared among related classes, when appropriate.** There are many cases in which a variety of related objects must all exhibit the same behavior at the same time. As an example, consider a video game that must display the following classes: **Player, PlayerBody, PlayerArms, PlayerLegs, PlayerHead**. Each of these classes has its own attributes (e.g., **position, orientation, color, speed**) and all must be updated and displayed as the user manipulates a joystick. The responsibilities *update()* and *display()* must therefore be shared by each of the objects noted. **Player** knows when something has changed and *update()* is required. It collaborates with the other objects to achieve a new position or orientation, but each object controls its own display.

**Collaborations.** Classes fulfill their responsibilities in one of two ways: (1) a class can use its own operations to manipulate its own attributes, thereby fulfilling a particular responsibility, or (2) a class can collaborate with other classes.

---

<sup>23</sup> In such cases, it may be necessary to split the class into multiple classes or complete subsystems in order to distribute intelligence more effectively.

Wirfs-Brock and her colleagues [WIR90] define *collaborations* in the following way:

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

Collaborations identify relationships between classes. When a set of classes all collaborate to achieve some requirement, they can be organized into a subsystem (a design issue).

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

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

To help in the identification of collaborators, the analyst can examine three different generic relationships between classes [WIR90]: (1) the *is-part-of* relationship, (2) the *has-knowledge-of* relationship, and (3) the *depends-upon* relationship. Each of the three generic relationships is considered briefly in the paragraphs that follow.

All classes that are part of an aggregate class are connected to the aggregate class via an *is-part-of* relationship. Consider the classes defined for the video game noted earlier, the class **PlayerBody** *is-part-of* **Player**, as are **PlayerArms**, **PlayerLegs**, and **PlayerHead**. In UML, these relationships are represented as the aggregation shown in Figure 8.16.

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

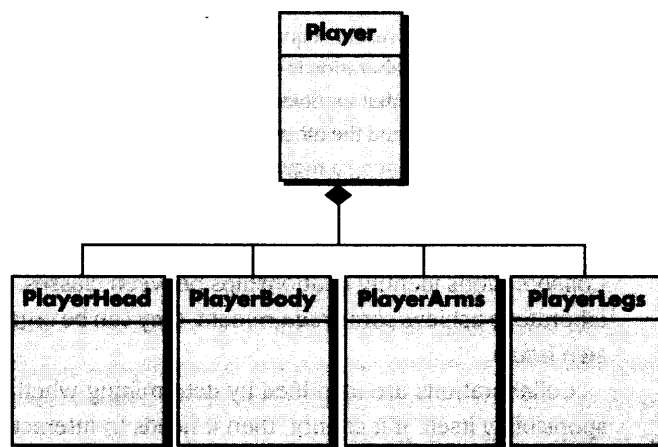
The *depends-upon* relationship implies that two classes have a dependency that is not achieved by *has-knowledge-of* or *is-part-of*. For example, **PlayerHead** must always be connected to **PlayerBody** (unless the video game is particularly violent), yet each object could exist without direct knowledge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

## KEY POINT

If a class cannot fulfill all of its obligations itself, then a collaboration is required.

FIGURE 8.16

A composite  
aggregate  
class



In all cases, the collaborator class name is recorded on the CRC model index card next to the responsibility that has spawned the collaboration. Therefore, the index card contains a list of responsibilities and the corresponding collaborations that enable the responsibilities to be fulfilled (Figure 8.15).

When a complete CRC model has been developed, representatives from the customer and software engineering organizations can review the model using the following approach [AMB95]:

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

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

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

index card is the collaborator **Sensor**. The token is then passed to the **Sensor** class.

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

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

## SAFEHOME



### CRC models

**The scene:** Ed's cubicle, as analysis modeling continues.

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

#### The conversation:

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

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

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

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

**Main cases:** SafeHome home management function.

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

visual devices—audio, television, DVD, digital recorders, and so forth.

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

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

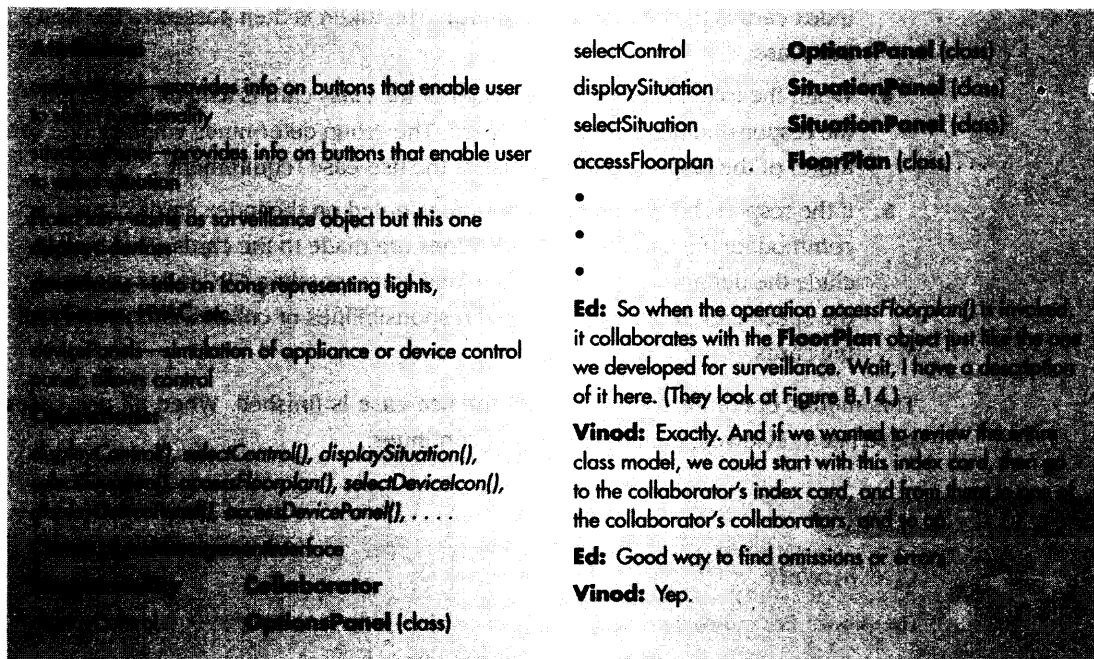
**Vinod (smiling):** They're working on it, say it's no biggy. Anyway, I extracted a bunch of classes for home management, and we can use one as an example. Let's use the **HomeManagementInterface** class.

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

**Vinod:** I thought you didn't understand CRC.

**Ed:** Maybe a little, but go ahead.

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



## KEY POINT

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

### 8.7.5 Associations and Dependencies

In many instances, two analysis classes are related to one another in some fashion, much like two data objects may be related to one another (Section 8.3.3). In UML these relationships are called *associations*. Referring back to Figure 8.14, the **FloorPlan** class is defined by identifying a set of associations between **FloorPlan** and two other classes, **Camera** and **Wall**. The class **Wall** is associated with three classes that allow a wall to be constructed, **WallSegment**, **Window**, and **Door**.

In some cases, an association may be further defined by indicating *multiplicity* (the term *cardinality* was used earlier in this chapter). Referring to Figure 8.14, a **Wall** object is constructed from one or more **WallSegment** objects. In addition, the **Wall** object may contain 0 or more **Window** objects and 0 or more **Door** objects. These multiplicity constraints are illustrated in Figure 8.17, where “one or more” is represented using  $1..*$ , and “0 or more” by  $0..*$ . In UML, the asterisk indicates an unlimited upper bound on the range.<sup>24</sup>

In many instances, a client-server relationship exists between two analysis classes. In such cases, a client-class depends on the server-class in some way and a *dependency relationship* is established. Dependencies are defined by a stereotype. A *stereotype* is an “extensibility mechanism” [ARL02] within UML that allows a software

## What is a stereotype?

<sup>24</sup> Other multiplicity relations—one to one, one to many, many to many, one to a specified range with lower and upper limits, and others—may be indicated as part of an association.



FIGURE 8.17

Multiplicity

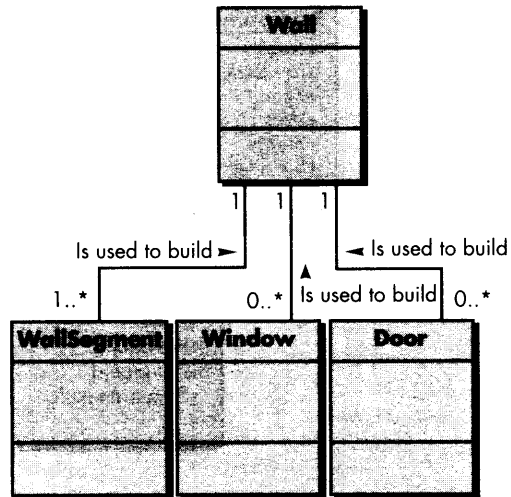
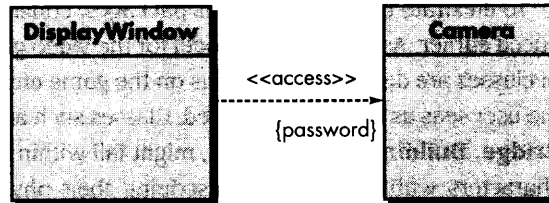


FIGURE 8.18

Dependencies



engineer to define a special modeling element whose semantics are custom-defined. In UML stereotypes are represented in double angle brackets (e.g., «stereotype»).

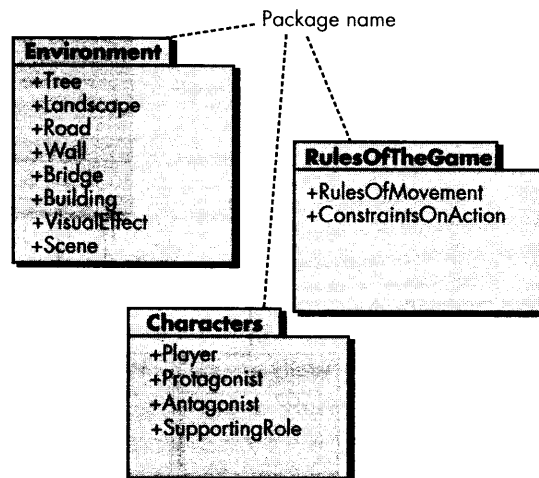
As an illustration of a simple dependency within the *SafeHome* surveillance system, a **Camera** object (in this case, the server-class) provides a video image to a **DisplayWindow** object (in this case, the client-class). The relationship between these two objects is not a simple association, yet a dependency association does exist. In a use-case written for surveillance (not shown), the modeler learns that a special password must be provided in order to view specific camera locations. One way to achieve this is to have **Camera** request a password and then grant permission to the **DisplayWindow** to produce the video display. This can be represented as shown in Figure 8.18 where «access» implies that the use of the camera output is controlled by a special password.

### 8.7.6 Analysis Packages

An important part of analysis modeling is categorization. That is, various elements of the analysis model (e.g., use-cases, analysis classes) are categorized in a manner

FIGURE 8.19

Packages



that packages them as a grouping—called an *analysis package*—that is given a representative name.

### KEY POINT

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

To illustrate the use of analysis packages, consider the video game that we introduced earlier. As the analysis model for the video game is developed, a large number of classes are derived. Some focus on the game environment—the visual scenes that the user sees as the game is played. Classes such as **Tree**, **Landscape**, **Road**, **Wall**, **Bridge**, **Building**, **VisualEffect**, might fall within this category. Others focus on the characters within the game, describing their physical features, actions, and constraints. Classes such as **Player** (described earlier), **Protagonist**, **Antagonist**, **SupportingRoles**, might be defined. Still others describe the rules of the game—how a player navigates through the environment. Classes such as **Rules OfMovement** and **ConstraintsOnAction** are candidates here. Many other categories might exist. These classes can be represented as analysis packages as shown in Figure 8.19.

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

## 8.8 CREATING A BEHAVIORAL MODEL

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

Class diagrams, CRC index cards, and other class-oriented models discussed in Section 8.7 represent static elements of the analysis model. It is now time to make a transition to the dynamic behavior of the system or product. To accomplish this, we must represent the behavior of the system as a function of specific events and time.

The *behavioral model* indicates how software will respond to external events or stimuli. To create the model, the analyst must perform the following steps:

### KEY POINT

Use-cases are parsed to define events. To accomplish this, the use-case is examined for points of information exchange.

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 classes.
3. Create a sequence for each use-case.
4. Build a state diagram for the system.
5. Review the behavioral model to verify accuracy and consistency.

Each of these steps is discussed in the sections that follow.

#### 8.8.1 Identifying Events with the Use-Case

As we noted in Section 8.5, 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. Recalling our earlier discussion of behavioral modeling in Section 8.6.3, it is important to note that 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, we reconsider the use-case for a small 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 analysis model, the object, **Homeowner**,<sup>25</sup> 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* does not explicitly change the flow of control of the use-case, but the results of the event *compare password* (derived from the interaction “password is compared with the valid password

<sup>25</sup> In this example, we assume that each user (homeowner) that interacts with *SafeHome* has an identifying password and is therefore a legitimate object.

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 *compare password* event).

### 8.8.2 State Representations

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.<sup>26</sup>

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 earlier) 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 that represents active states for each class and the events (triggers) that cause changes between these active states. Figure 8.20 illustrates a state diagram for the **ControlPanel** class in the *SafeHome* security function.

Each arrow shown in Figure 8.20 represents a transition from one active state of a class 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 a class, it is possible to specify additional information to provide more depth in understanding the behavior of a class. In addition to specifying the event that causes the transition to occur, the analyst can specify a guard and an action [CHA93]. A *guard* is a Boolean condition that must be satisfied in order for the

#### 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.

<sup>26</sup> The state diagrams presented in Section 8.6.3 depict the state of the system. Our discussion in this section will focus on the state of each class within the analysis model.