CHAPTER

10 Requirements Modeling: Class-Based Methods

Key Concepts

When they were first introduced in the early 1990s, class-based methods for requirements modeling were often categorized as *object-oriented analysis*. Although a number of different class-based methods and representations were introduced, Coad and Yourdon [Coa91] noted one universal characteristic for all of them:

> [Object-oriented methods are all] based upon concepts that we first learned in kindergarten: objects and attributes, wholes and parts, classes and members.

Class-based methods for requirements modeling use these common concepts to craft a representation of an application that can be understood by nontechnical stakeholders. As the requirements model is refined and expanding, it evolves into a specification that can be used by software engineers in the creation of the software design.

Class-based modeling represents the objects that the system will manipulate, the operations (also called *methods* or *services*) that will be applied to the objects to effect the manipulation, relationships (some hierarchical) between the objects, and the collaborations that occur between the classes that are

Quick Look

What is it? Software problems can almost always be characterized in terms of a set of interacting objects each representing something of interest within a system. Each object becomes a member of a class of objects. Each object is described by its state—the data attributes that describe the object. All of this can be represented using class-based requirements modeling methods.

Who does it? A software engineer (sometimes called an analyst) builds the class-based model using requirements elicited from the customer.

Why is it important? A class-based requirements model makes use of objects drawn from the customer's view of an application or system. The model depicts a view of the system that is common to the customer. Therefore, it can be readily evaluated by the customer, resulting in useful feedback at the earliest possible time. Later, as the model is refined, it becomes the basis for software design.

What are the steps? Class-based modeling defines objects, attributes, and relationships. A set of simple heuristics can be developed to extract objects and classes from a problem statement and then represent them in text-based and/or diagrammatic forms. Once preliminary models are created, they are refined and analyzed to assess their clarity, completeness, and consistency.

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

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

defined. The elements of a class-based model include classes and objects, attributes, operations, class-responsibility-collaborator (CRC) models, collaboration diagrams, and packages. The sections that follow present a series of informal guidelines that will assist in their identification and representation.

10.1 Identifying Analysis Classes

Quote:

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

Carl Argila

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

We can begin to identify classes by examining the usage scenarios developed as part of the requirements model (Chapter 9) and performing a "grammatical parse" [Abb83] on the use cases developed for the system to be built. Classes are determined by underlining each noun or noun phrase and entering it into a simple table. Synonyms should be noted. If the class (noun) is required to implement a solution, then it is part of the solution space; otherwise, if a class is necessary only to describe a solution, it is part of the problem space.

But what should we look for once all 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.[1] For example, Budd [Bud96] suggests a taxonomy of classes that includes

1 Another important categorization, defining entity, boundary, and controller classes, is discussed in Section 10.5.

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 object. It is likely that inversion would be defined as an operation for the object **Image,** but it would not be defined as a separate class to connote "image inversion." As Cashman [Cas89] states, "[T]he 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, consider a grammatical parse (nouns are underlined, verbs italicized) for a processing narrative[2] for the *SafeHome* security function.

ADVICE

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.

> The <u>SafeHome security function</u> *enables* the <u>homeowner</u> to *configure* the <u>security system</u> when it is *installed, monitors* all <u>sensors</u> *connected* to the security system, and *interacts* with the homeowner through the <u>Internet</u>, a <u>PC</u> or a <u>control panel</u>.
>
> During <u>installation</u>, the SafeHome PC is used to *program* and *configure* the <u>system</u>. Each sensor is assigned a <u>number</u> and <u>type</u>, a <u>master password</u> is programmed for *arming* and *disarming* the system, and <u>telephone number(s)</u> are *input* for *dialing* when a <u>sensor event</u> occurs.
>
> When a sensor event is *recognized*, the software *invokes* an <u>audible alarm</u> attached to the system. After a <u>delay time</u> that is *specified* by the homeowner during system configuration activities, the software dials a telephone number of a <u>monitoring service</u>, *provides* <u>information</u> about the <u>location</u>, *reporting* the nature of the event that has been detected. The telephone number will be *redialed* every 20 seconds until <u>telephone connection</u> is *obtained.*
>
> The homeowner *receives* <u>security information</u> via a control panel, the PC, or a browser, collectively called an <u>interface</u>. The interface *displays* <u>prompting messages</u> and <u>system status information</u> on the control panel, the PC, or the browser window. Homeowner interaction takes the following form . . .

2 A processing narrative is similar to the use case in style but somewhat different in purpose. The processing narrative provides an overall description of the function to be developed. It is not a scenario written from one actor's point of view. It is important to note, however, that a grammatical parse can also be used for every use case developed as part of requirements gathering (elicitation).

Extracting the nouns, we can propose 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.

Coad and Yourdon [Coa91] suggest six selection characteristics that should be used as you consider each potential class for inclusion in the analysis model:

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

1. *Retained information*. The potential class will be useful during analysis only if information about it must be remembered so that the system can function.
2. *Needed services*. The potential class must have a set of identifiable operations that can change the value of its attributes in some way.
3. *Multiple attributes*. During requirement analysis, the focus should be on "major" information; a class with a single attribute may, in fact, be useful during design, but is probably better represented as an attribute of another class during the analysis activity.
4. *Common attributes*. A set of attributes can be defined for the potential class and these attributes apply to all instances of the class.
5. *Common operations*. A set of operations can be defined for the potential class and these operations apply to all instances of the class.
6. *Essential requirements*. External entities that appear in the problem space and produce or consume information essential to the operation of any solution for the system will almost always be defined as classes in the requirements model.

Quote:

"Classes struggle, some classes triumph, others are eliminated."

Mao Zedong

To be considered a legitimate class for inclusion in the requirements model, a potential object 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 an object to be discarded or reinstated.

However, the first step of class-based modeling is the definition of classes, and decisions (even subjective ones) must be made. With this in mind, you should apply the selection characteristics to the list of potential *SafeHome* classes:

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

It should be noted that (1) the preceding list is not all inclusive, 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).

10.2 Specifying Attributes

KEY POINT

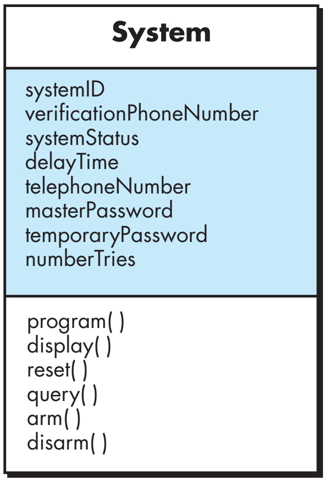
Attributes are the set of data objects that fully define the class within the context of the problem.

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. For example, if we were to build a system that tracks baseball statistics for professional baseball players, the attributes of the class **Player** would be quite different than the attributes of the same class when it is used in the context of the professional baseball pension system. In the former, attributes such as name, position, batting average, fielding percentage, years played, and games played might be relevant. For the latter, some of these attributes would be meaningful, but others would be replaced (or augmented) by attributes like average salary, credit toward full vesting, pension plan options chosen, mailing address, and the like.

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

FIGURE 10.1

Class diagram for the system class



To illustrate, we consider the **System** class defined for *SafeHome*. A homeowner can configure the security function to reflect sensor information, alarm response information, activation/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

Each of the data items to the right of the equal sign could be further defined to an elementary level, but for our purposes, they constitute a reasonable list of attributes for the **System** class (shaded portion of Figure 10.1).

Sensors are part of the overall *SafeHome* system, and yet they are not listed as data items or as attributes in Figure 10.1. **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.

10.3 DEFINING OPERATIONS

Operations define the behavior of an object. Although many different types of operations exist, they can generally be divided into four broad categories: (1) operations that manipulate data in some way (e.g., adding, deleting, reformatting, selecting), (2) operations that perform a computation, (3) operations that inquire about the state of an object, and (4) operations that monitor an object for the occurrence of a controlling event. These functions are accomplished by

ADVICE

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

operating on attributes and/or associations (Section 10.5). 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, you can again study a processing narrative (or use case) and select those operations that reasonably belong to the class. To accomplish this, the grammatical parse is again studied and verbs are isolated. Some of these verbs will be legitimate operations and can be easily connected to a specific class. For example, from the *SafeHome* processing narrative presented earlier in this chapter, we see that "sensor is *assigned* a number and type" or "a master password is *programmed* for *arming and disarming* the system." These phrases indicate a number of things:

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

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.

In addition to the grammatical parse, you can gain additional insight into other operations by considering the communication that occurs between objects. Objects communicate by passing messages to one another. Before continuing with the specification of operations, we explore this matter in a bit more detail.

SAFEHOME

Class Models

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

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

The conversation:

[Ed has been working to extract classes from the use case template for ACS-DCV (presented in an earlier sidebar in this chapter) and is presenting the classes he has extracted to his colleagues.]

Ed: So when the homeowner wants to pick a camera, he or she has to pick it from a floor plan. I've defined a **FloorPlan** class. Here's the diagram.

(They look at Figure 10.2.)

Jamie: So **FloorPlan** is an object that is put together with walls, doors, windows, and cameras. That's what those labeled lines mean, right?

Ed: Yeah, they're called "associations." One class is associated with another according to the associations I've shown. [Associations are discussed in Section 10.5.]

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 attributes under, say, **WallSegment,** which is used to build a wall. The wall segment has start and stop coordinates and the *draw()* operation does the rest.

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

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

Vinod: I have a question. Why does the camera have an ID but the others don't? I notice you have an attribute called **nextWall**. How will **WallSegment** know what the next wall will be?

Ed: Good question, but as they say, that's a design decision, so I'm going to delay that until . . .

Jamie: Give me a break . . . I'll bet you've already figured it out.

Ed (smiling sheepishly): True, I'm gonna use a list structure which I'll model when we get to design. If you get religious about separating analysis and design, the level of detail I have right here could be suspect.

Jamie: Looks pretty good to me, but I have a few more questions.
(Jamie asks questions which result in minor modifications.)

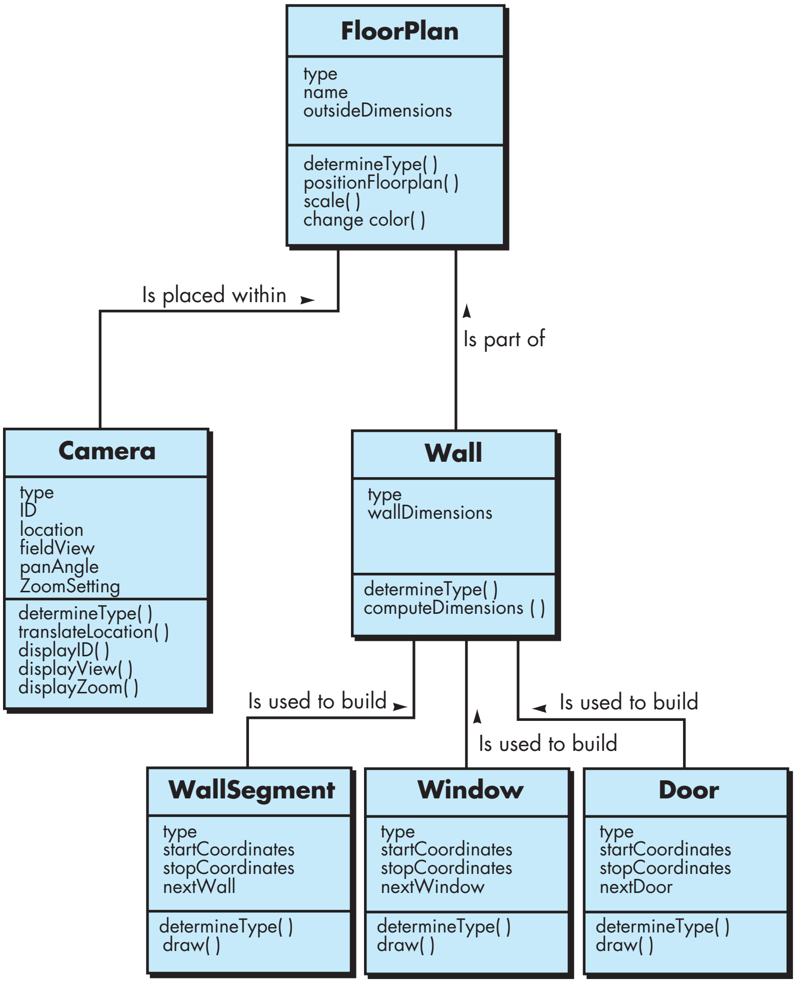
Vinod: Do you have CRC cards for each of the objects? If so, we ought to role-play through them, just to make sure nothing has been omitted.

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

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

FIGURE 10.2

Class diagram for FloorPlan (see sidebar discussion)



10.4 Class-Responsibility-Collaborator Modeling

Quote:

"One 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

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.

WebRef

An excellent discussion of these class types can be found at **http://www.oracle.com/technetwork/developer-tools/jdev/gettingstartedwithumlclassmodeling-130316.pdf.**

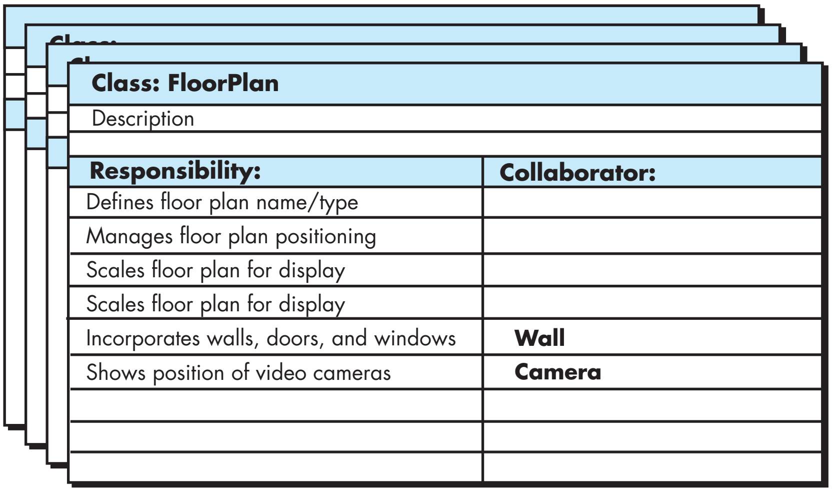
A simple CRC index card for the **FloorPlan** class is illustrated in Figure 10.3. The list of responsibilities shown on the CRC card is preliminary and subject to additions or modification. The classes **Wall** and **Camera** are noted next to the responsibility that will require their collaboration.

Classes. Basic guidelines for identifying classes and objects were presented earlier in this chapter. The taxonomy of class types presented in Section 10.1 can be extended by considering the following categories:

- *Entity classes*, also called *model* or *business* classes, are extracted directly from the statement of the problem (e.g., **FloorPlan** and **Sensor**).

Figure 10.3

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	**Wall**
Shows position of video cameras	**Camera**

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).

Quote:

"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

- *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 objects 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" 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, (4) validation of data communicated between objects or between the user and the application. In general, controller classes are not considered until the design activity has begun.

? What guidelines can be applied for allocating responsibilities to classes?

Responsibilities. Basic guidelines for identifying responsibilities (attributes and operations) have been presented in Sections 10.2 and 10.3. Wirfs-Brock and her colleagues [Wir90] suggest five guidelines 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: it concentrates all intelligence within a few classes, making changes more difficult, and 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), the cohesiveness of the system is improved.[3] This enhances the maintainability of the software and reduces the impact of side effects due to change.

3 Cohesiveness is a design concept that is discussed in Chapter 12.

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.[4] In addition, the responsibilities for each class should exhibit the same level of abstraction. For example, among the operations listed for an aggregate class called **CheckingAccount** a reviewer notes two responsibilities: *balance-the-account* and *check-off-cleared-checks*. The first operation (responsibility) implies a reasonably complex mathematical and logical procedure. The second is a simple clerical activity. Since these two operations are not at the same level of abstraction, *check-off-cleared-checks* should be placed within the responsibilities of **CheckEntry,** a class that is encompassed by the aggregate class **CheckingAccount.**

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 object-oriented 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.

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

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. 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 is associated with a particular responsibility implemented by the server.

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, you 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.

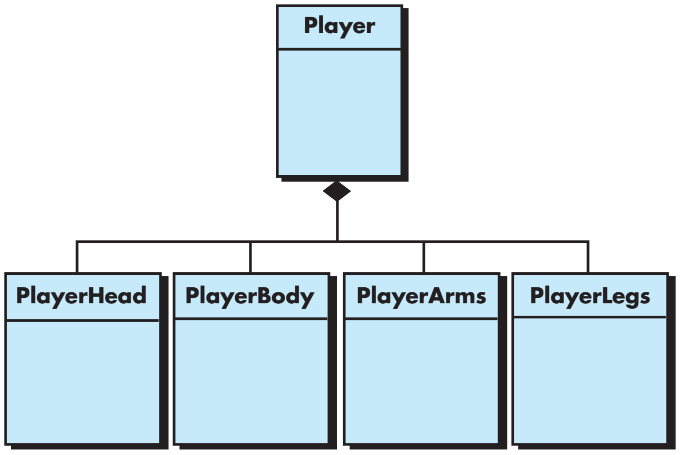
All classes that are part of an aggregate class are connected to the aggregate class via an *is-part-of* relationship. Consider the classes defined for the video game noted earlier, the class **PlayerBody** *is-part-of* **Player**, as are **PlayerArms, PlayerLegs,** and **PlayerHead.** In UML, these relationships are represented as the aggregation shown in Figure 10.4.

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.**

FIGURE 10.4

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 10.3).

When a complete CRC model has been developed, the representatives from the stakeholders can review the model using the following approach [Amb95]:

1. All participants in the review (of the CRC model) are given a subset of the CRC model index cards. 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 object, 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 contains 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** object.

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 (or use case diagrams) have been reviewed, requirements modeling continues.

SAFEHOME

CRC Models

The scene: Ed's cubicle, as requirements modeling begins.

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

The conversation:
[Vinod has decided to show Ed how to develop CRC cards by showing him an example.]

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

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

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

Use case: *SafeHome* home management function.

Narrative: We want to use the home management interface on a 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 audiovisual devices—audio, television, DVD, digital recorders, and so forth.

With a single selection, I want to be able to set the entire house for various situations. One is home, another is away, a third is overnight travel, and a fourth is extended travel. All 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 setting 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 problem. Anyway, I extracted a bunch of classes for home management and we can use one as an example. Let's use the **HomeManagementInterface** class.

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

Vinod: I thought you didn't understand CRC.

Ed: Maybe a little, but go ahead.

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

Attributes:

optionsPanel—contains info on buttons that enable user to select functionality.

situationPanel—contains info on buttons that enable user to select situation.

floorplan—same as surveillance object but this one displays devices.

deviceIcons—info on icons representing lights, appliances, HVAC, etc.

devicePanels—simulation of appliance or device control panel; allows control.

Operations:
displayControl(), selectControl(), displaySituation(), select situation(), accessFloorplan(), selectDeviceIcon(), displayDevicePanel(), accessDevicePanel(), . . .

Class: HomeManagementInterface

Responsibility	Collaborator
displayControl()	**OptionsPanel** (class)
selectControl()	**OptionsPanel** (class)
displaySituation()	**SituationPanel** (class)
selectSituation()	**SituationPanel** (class)
accessFloorplan()	**FloorPlan** (class) . . .
. . .	

Ed: So when the operation *accessFloorplan()* is invoked, it collaborates with the **FloorPlan** object just like the one we developed for surveillance. Wait, I have a description of it here. (They look at Figure 10.2.)

Vinod: Exactly. And if we wanted to review the entire class model, we could start with this index card, then go to the collaborator's index card, and from there to one of the collaborator's collaborators, and so on.

Ed: Good way to find omissions or errors.

Vinod: Yep.

10.5 Associations and Dependencies

In many instances, two analysis classes are related to one another in some fashion. In UML these relationships are called *associations*. Referring back to Figure 10.2, 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**.

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

In some cases, an association may be further defined by indicating *multiplicity*. Referring to Figure 10.2, 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 10.5, 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.[5]

What is a stereotype?

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 you 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

5 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 10.5

Multiplicity

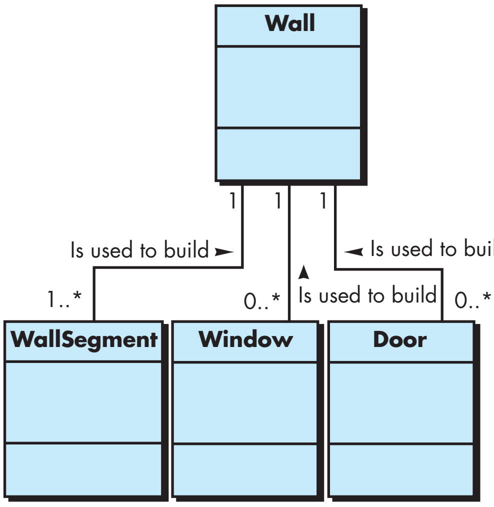
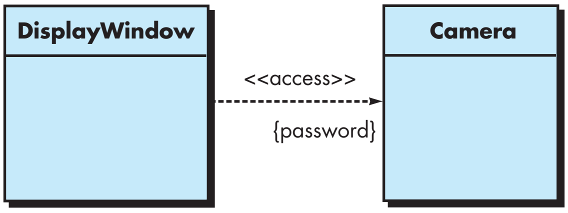


FIGURE 10.6

Dependencies



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), you learn 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 10.6 where <<access>> implies that the use of the camera output is controlled by a special password.

10.6 Analysis Packages

KEY POINT

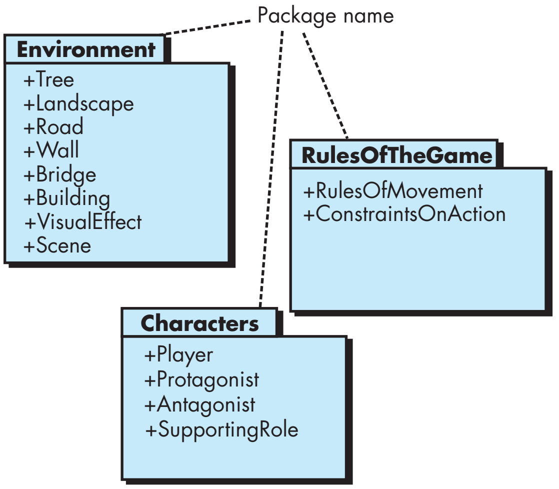
A package is used to assemble a collection of related classes.

An important part of analysis modeling is categorization. That is, various elements of the requirements model (e.g., use cases, analysis classes) are categorized in a manner that packages them as a grouping—called an *analysis package*—that is given a representative name.

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

FIGURE 10.7

Packages



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,** and **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,** and **SupportingRoles** might be defined. Still others describe the rules of the game—how a player navigates through the environment. Classes such as **RulesOfMovement** and **ConstraintsOnAction** are candidates here. Many other categories might exist. These classes can be represented as analysis classes as shown in Figure 10.7.

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 packages contained within a given package.

10.7 Summary

Class-based modeling uses information derived from use cases and other written application descriptions to identify analysis classes. A grammatical parse may be used to extract candidate classes, attributes, and operations from text-based narratives. Criteria for the definition of a class are defined.

A set of class-responsibility-collaborator index cards can be used to define relationships between classes. In addition, a variety of UML modeling notation can be applied to define hierarchies, relationships, associations, aggregations,