CHAPTER

5

UNDERSTANDING REQUIREMENTS

Understanding the requirements of a problem is among the most difficult tasks that face a software engineer. When you first think about it, developing a clear understanding of requirements doesn't seem that hard. After all, doesn't the customer know what is required? Shouldn't the end users have a good understanding of the features and functions that will provide benefit? Surprisingly, in many instances the answer to these questions is "no." And even if customers and end-users are explicit in their needs, those needs will change throughout the project.

In the forward to a book by Ralph Young [You01] on effective requirements practices, I wrote:

> It's your worst nightmare. A customer walks into your office, sits down, looks you straight in the eye, and says, "I know you think you understand what I said, but what you don't understand is what I said is not what I meant." Invariably, this happens late

QUICK LOOK

What is it? Before you begin any technical work, it's a good idea to apply a set of requirements engineering tasks. These tasks lead to an understanding of what the business impact of the software will be, what the customer wants, and how end users will interact with the software.

Who does it? Software engineers (sometimes referred to as system engineers or "analysts" in the IT world) and other project stakeholders (managers, customers, end users) all participate in requirements engineering.

Why is it important? Designing and building an elegant computer program that solves the wrong problem serves no one's needs. That's why it's important to understand what the customer wants before you begin to design and build a computer-based system.

What are the steps? Requirements engineering begins with inception—a task that defines the scope and nature of the problem to be solved. It moves onwards to elicitation—a task that helps stakeholders define what is required, and then

elaboration—where basic requirements are refined and modified. As stakeholders define the problem, negotiation occurs—what are the priorities, what is essential, when is it required? Finally, the problem is specified in some manner and then reviewed or validated to ensure that your understanding of the problem and the stakeholders' understanding of the problem coincide.

What is the work product? The intent of requirements engineering is to provide all parties with a written understanding of the problem. This can be achieved though a number of work products: usage scenarios, functions and features lists, requirements models, or a specification.

How do I ensure that I've done it right? Requirements engineering work products are reviewed with stakeholders to ensure that what you have learned is what they really meant. A word of warning: even after all parties agree, things will change, and they will continue to change throughout the project.

in the project, after deadline commitments have been made, reputations are on the line, and serious money is at stake.

All of us who have worked in the systems and software business for more than a few years have lived this nightmare, and yet, few of us have learned to make it go away. We struggle when we try to elicit requirements from our customers. We have trouble understanding the information that we do acquire. We often record requirements in a disorganized manner, and we spend far too little time verifying what we do record. We allow change to control us, rather than establishing mechanisms to control change. In short, we fail to establish a solid foundation for the system or software. Each of these problems is challenging. When they are combined, the outlook is daunting for even the most experienced managers and practitioners. But solutions do exist.

It's reasonable to argue that the techniques I'll discuss in this chapter are not a true "solution" to the challenges just noted. But they do provide a solid approach for addressing these challenges.

5.1   REQUIREMENTS ENGINEERING

Quote:

"The hardest single part of building a software system is deciding what to build. No part of the work so cripples the resulting system if done wrong. No other part is more difficult to rectify later."

Fred Brooks

KEY POINT

Requirements engineering establishes a solid base for design and construction. Without it, the resulting software has a high probability of not meeting customer's needs.

Designing and building computer software is challenging, creative, and just plain fun. In fact, building software is so compelling that many software developers want to jump right in before they have a clear understanding of what is needed. They argue that things will become clear as they build, that project stakeholders will be able to understand need only after examining early iterations of the software, that things change so rapidly that any attempt to understand requirements in detail is a waste of time, that the bottom line is producing a working program and all else is secondary. What makes these arguments seductive is that they contain elements of truth.[1] But each is flawed and can lead to a failed software project.

The broad spectrum of tasks and techniques that lead to an understanding of requirements is called *requirements engineering.* From a software process perspective, requirements engineering is a major software engineering action that begins during the communication activity and continues into the modeling activity. It must be adapted to the needs of the process, the project, the product, and the people doing the work.

Requirements engineering builds a bridge to design and construction. But where does the bridge originate? One could argue that it begins at the feet of the project stakeholders (e.g., managers, customers, end users), where business need is defined, user scenarios are described, functions and features are delineated, and project constraints are identified. Others might suggest that it begins with a broader system definition, where software is but one component of the larger system domain. But regardless of the starting point, the journey across the bridge takes you

---

1   This is particularly true for small projects (less than one month) and smaller, relatively simple software efforts. As software grows in size and complexity, these arguments begin to break down.

high above the project, allowing you to examine the context of the software work to be performed; the specific needs that design and construction must address; the priorities that guide the order in which work is to be completed; and the information, functions, and behaviors that will have a profound impact on the resultant design.

Requirements engineering provides the appropriate mechanism for understanding what the customer wants, analyzing need, assessing feasibility, negotiating a reasonable solution, specifying the solution unambiguously, validating the specification, and managing the requirements as they are transformed into an operational system [Tha97]. It encompasses seven distinct tasks: inception, elicitation, elaboration, negotiation, specification, validation, and management. It is important to note that some of these tasks occur in parallel and all are adapted to the needs of the project.

Inception.   How does a software project get started? Is there a single event that becomes the catalyst for a new computer-based system or product, or does the need evolve over time? There are no definitive answers to these questions. In some cases, a casual conversation is all that is needed to precipitate a major software engineering effort. But in general, most projects begin when a business need is identified or a potential new market or service is discovered. Stakeholders from the business community (e.g., business managers, marketing people, product managers) define a business case for the idea, try to identify the breadth and depth of the market, do a rough feasibility analysis, and identify a working description of the project's scope. All of this information is subject to change, but it is sufficient to precipitate discussions with the software engineering organization.[2]

At project inception,[3] you establish a basic understanding of the problem, the people who want a solution, the nature of the solution that is desired, and the effectiveness of preliminary communication and collaboration between the other stakeholders and the software team.

Elicitation.   It certainly seems simple enough—ask the customer, the users, and others what the objectives for the system or product are, what is to be accomplished, how the system or product fits into the needs of the business, and finally, how the system or product is to be used on a day-to-day basis. But it isn't simple—it's very hard.

Christel and Kang [Cri92] identify a number of problems that are encountered as elicitation occurs.

- **Problems of scope.** The boundary of the system is ill-defined or the customers/users specify unnecessary technical detail that may confuse, rather than clarify, overall system objectives.

---

2   If a computer-based system is to be developed, discussions begin within the context of a system engineering process. For a detailed discussion of system engineering, visit the website that accompanies this book.
3   Recall that the Unified Process (Chapter 2) defines a more comprehensive "inception phase" that encompasses the inception, elicitation, and elaboration tasks discussed in this chapter.

- **Problems of understanding.** The customers/users are not completely sure of what is needed, have a poor understanding of the capabilities and limitations of their computing environment, don't have a full understanding of the problem domain, have trouble communicating needs to the system engineer, omit information that is believed to be "obvious," specify requirements that conflict with the needs of other customers/users, or specify requirements that are ambiguous or untestable.

- **Problems of volatility.** The requirements change over time.

To help overcome these problems, you must approach requirements gathering in an organized manner.

Elaboration.   The information obtained from the customer during inception and elicitation is expanded and refined during elaboration. This task focuses on developing a refined requirements model (Chapters 6 and 7) that identifies various aspects of software function, behavior, and information.

Elaboration is driven by the creation and refinement of user scenarios that describe how the end user (and other actors) will interact with the system. Each user scenario is parsed to extract analysis classes—business domain entities that are visible to the end user. The attributes of each analysis class are defined, and the services[4] that are required by each class are identified. The relationships and collaboration between classes are identified, and a variety of supplementary diagrams are produced.

Negotiation.   It isn't unusual for customers and users to ask for more than can be achieved, given limited business resources. It's also relatively common for different customers or users to propose conflicting requirements, arguing that their version is "essential for our special needs."

You have to reconcile these conflicts through a process of negotiation. Customers, users, and other stakeholders are asked to rank requirements and then discuss conflicts in priority. Using an iterative approach that prioritizes requirements, assesses their cost and risk, and addresses internal conflicts, requirements are eliminated, combined, and/or modified so that each party achieves some measure of satisfaction.

Specification.   In the context of computer-based systems (and software), the term *specification* means different things to different people. A specification can be a written document, a set of graphical models, a formal mathematical model, a collection of usage scenarios, a prototype, or any combination of these.

Some suggest that a "standard template" [Som97] should be developed and used for a specification, arguing that this leads to requirements that are presented in a

ADVICE

Elaboration is a good thing, but you have to know when to stop. The key is to describe the problem in a way that establishes a firm base for design. If you work beyond that point, you're doing design.

ADVICE

There should be no winner and no loser in an effective negotiation. Both sides win, because a "deal" that both can live with is solidified.

---

4   A *service* manipulates the data encapsulated by the class. The terms *operation* and *method* are also used. If you are unfamiliar with object-oriented concepts, a basic introduction is presented in Appendix 2.

consistent and therefore more understandable manner. However, it is sometimes necessary to remain flexible when a specification is to be developed. For large systems, a written document, combining natural language descriptions and graphical models may be the best approach. However, usage scenarios may be all that are required for smaller products or systems that reside within well-understood technical environments.

INFO

Software Requirements Specification Template

A *software requirements specification* (SRS) is a document that is created when a detailed description of all aspects of the software to be built must be specified before the project is to commence. It is important to note that a formal SRS is not always written. In fact, there are many instances in which effort expended on an SRS might be better spent in other software engineering activities. However, when software is to be developed by a third party, when a lack of specification would create severe business issues, or when a system is extremely complex or business critical, an SRS may be justified.

Karl Wiegers [Wie03] of Process Impact Inc. has developed a worthwhile template (available at **www.processimpact.com/process_assets/srs_template.doc**) that can serve as a guideline for those who must create a complete SRS. A topic outline follows:

Table of Contents
Revision History

1.   Introduction
1.1   Purpose
1.2   Document Conventions
1.3   Intended Audience and Reading Suggestions
1.4   Project Scope
1.5   References

2.   Overall Description
2.1   Product Perspective

2.2   Product Features
2.3   User Classes and Characteristics
2.4   Operating Environment
2.5   Design and Implementation Constraints
2.6   User Documentation
2.7   Assumptions and Dependencies

3.   System Features
3.1   System Feature 1
3.2   System Feature 2 (and so on)

4.   External Interface Requirements
4.1   User Interfaces
4.2   Hardware Interfaces
4.3   Software Interfaces
4.4   Communications Interfaces

5.   Other Nonfunctional Requirements
5.1   Performance Requirements
5.2   Safety Requirements
5.3   Security Requirements
5.4   Software Quality Attributes

6.   Other Requirements

Appendix A: Glossary
Appendix B: Analysis Models
Appendix C: Issues List

A detailed description of each SRS topic can be obtained by downloading the SRS template at the URL noted earlier in this sidebar.

Validation.   The work products produced as a consequence of requirements engineering are assessed for quality during a validation step. Requirements validation examines the specification[5] to ensure that all software requirements have been

---

5   Recall that the nature of the specification will vary with each project. In some cases, the "specification" is a collection of user scenarios and little else. In others, the specification may be a document that contains scenarios, models, and written descriptions.

stated unambiguously; that inconsistencies, omissions, and errors have been detected and corrected; and that the work products conform to the standards established for the process, the project, and the product.

The primary requirements validation mechanism is the technical review (Chapter 15). The review team that validates requirements includes software engineers, customers, users, and other stakeholders who examine the specification looking for errors in content or interpretation, areas where clarification may be required, missing information, inconsistencies (a major problem when large products or systems are engineered), conflicting requirements, or unrealistic (unachievable) requirements.

Requirements Validation Checklist

It is often useful to examine each requirement against a set of checklist questions. Here is a small subset of those that might be asked:

- Are requirements stated clearly? Can they be misinterpreted?
- Is the source (e.g., a person, a regulation, a document) of the requirement identified? Has the final statement of the requirement been examined by or against the original source?
- Is the requirement bounded in quantitative terms?
- What other requirements relate to this requirement? Are they clearly noted via a cross-reference matrix or other mechanism?

- Does the requirement violate any system domain constraints?
- Is the requirement testable? If so, can we specify tests (sometimes called validation criteria) to exercise the requirement?
- Is the requirement traceable to any system model that has been created?
- Is the requirement traceable to overall system/product objectives?
- Is the specification structured in a way that leads to easy understanding, easy reference, and easy translation into more technical work products?
- Has an index for the specification been created?
- Have requirements associated with performance, behavior, and operational characteristics been clearly stated? What requirements appear to be implicit?

Requirements management. Requirements for computer-based systems change, and the desire to change requirements persists throughout the life of the system. Requirements management is a set of activities that help the project team identify, control, and track requirements and changes to requirements at any time as the project proceeds.[6] Many of these activities are identical to the software configuration management (SCM) techniques discussed in Chapter 22.

---

6   Formal requirements management is initiated only for large projects that have hundreds of identifiable requirements. For small projects, this requirements engineering action is considerably less formal.

Requirements Engineering

Objective: Requirements engineering tools assist in requirements gathering, requirements modeling, requirements management, and requirements validation.

Mechanics: Tool mechanics vary. In general, requirements engineering tools build a variety of graphical (e.g., UML) models that depict the informational, functional, and behavioral aspects of a system. These models form the basis for all other activities in the software process.

Representative Tools:[7]
A reasonably comprehensive (and up-to-date) listing of requirements engineering tools can be found at the Volere Requirements resources site at **www.volere.co.uk/ tools.htm**. Requirements modeling tools are discussed in

Chapters 6 and 7. Tools noted below focus on requirement management.

EasyRM, developed by Cybernetic Intelligence GmbH (**www.easy-rm.com**), builds a project-specific dictionary/glossary that contains detailed requirements descriptions and attributes.

Rational RequisitePro, developed by Rational Software (**www-306.ibm.com/software/awdtools/ reqpro/**), allows users to build a requirements database; represent relationships among requirements; and organize, prioritize, and trace requirements.

Many additional requirements management tools can be found at the Volere site noted earlier and at **www.jiludwig.com/Requirements_ Management_Tools.html**.

5.2   ESTABLISHING THE GROUNDWORK

In an ideal setting, stakeholders and software engineers work together on the same team.[8] In such cases, requirements engineering is simply a matter of conducting meaningful conversations with colleagues who are well-known members of the team. But reality is often quite different.

Customer(s) or end users may be located in a different city or country, may have only a vague idea of what is required, may have conflicting opinions about the system to be built, may have limited technical knowledge, and may have limited time to interact with the requirements engineer. None of these things are desirable, but all are fairly common, and you are often forced to work within the constraints imposed by this situation.

In the sections that follow, I discuss the steps required to establish the groundwork for an understanding of software requirements—to get the project started in a way that will keep it moving forward toward a successful solution.

5.2.1   Identifying Stakeholders

Sommerville and Sawyer [Som97] define a stakeholder as "anyone who benefits in a direct or indirect way from the system which is being developed." I have already

KEY POINT

A *stakeholder* is anyone who has a direct interest in or benefits from the system that is to be developed.

---

7   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.
8   This approach is strongly recommended for projects that adopt an agile software development philosophy.

identified the usual suspects: business operations managers, product managers, marketing people, internal and external customers, end users, consultants, product engineers, software engineers, support and maintenance engineers, and others. Each stakeholder has a different view of the system, achieves different benefits when the system is successfully developed, and is open to different risks if the development effort should fail.

At inception, you should create a list of people who will contribute input as requirements are elicited (Section 5.3). The initial list will grow as stakeholders are contacted because every stakeholder will be asked: "Whom else do you think I should talk to?"

5.2.2 Recognizing Multiple Viewpoints

Because many different stakeholders exist, the requirements of the system will be explored from many different points of view. For example, the marketing group is interested in functions and features that will excite the potential market, making the new system easy to sell. Business managers are interested in a feature set that can be built within budget and that will be ready to meet defined market windows. End users may want features that are familiar to them and that are easy to learn and use. Software engineers may be concerned with functions that are invisible to nontechnical stakeholders but that enable an infrastructure that supports more marketable functions and features. Support engineers may focus on the maintainability of the software.

Each of these constituencies (and others) will contribute information to the requirements engineering process. As information from multiple viewpoints is collected, emerging requirements may be inconsistent or may conflict with one another. You should categorize all stakeholder information (including inconsistent and conflicting requirements) in a way that will allow decision makers to choose an internally consistent set of requirements for the system.

5.2.3 Working toward Collaboration

If five stakeholders are involved in a software project, you may have five (or more) different opinions about the proper set of requirements. Throughout earlier chapters, I have noted that customers (and other stakeholders) must collaborate among themselves (avoiding petty turf battles) and with software engineering practitioners if a successful system is to result. But how is this collaboration accomplished?

The job of a requirements engineer is to identify areas of commonality (i.e., requirements on which all stakeholders agree) and areas of conflict or inconsistency (i.e., requirements that are desired by one stakeholder but conflict with the needs of another stakeholder). It is, of course, the latter category that presents a challenge.

Quote:

"Put three stakeholders in a room and ask them what kind of system they want. You're likely to get four or more different opinions."

Author unknown

Using "Priority Points"

One way of resolving conflicting requirements and at the same time better understanding the relative importance of all requirements is to use a "voting" scheme based on *priority points.* All stakeholders are provided with some number of priority points that can be "spent" on any number of requirements. A list of requirements is presented, and each stakeholder indicates the relative importance of each (from his or her viewpoint) by spending one or more priority points on it. Points spent cannot be reused. Once a stakeholder's priority points are exhausted, no further action on requirements can be taken by that person. Overall points spent on each requirement by all stakeholders provide an indication of the overall importance of each requirement.

Collaboration does not necessarily mean that requirements are defined by committee. In many cases, stakeholders collaborate by providing their view of requirements, but a strong "project champion"(e.g., a business manager or a senior technologist) may make the final decision about which requirements make the cut.

5.2.4 Asking the First Questions

Questions asked at the inception of the project should be "context free" [Gau89]. The first set of context-free questions focuses on the customer and other stakeholders, the overall project goals and benefits. For example, you might ask:

Quote:

"It is better to know some of the questions than all of the answers."

James Thurber

- Who is behind the request for this work?
- Who will use the solution?
- What will be the economic benefit of a successful solution?
- Is there another source for the solution that you need?

These questions help to identify all stakeholders who will have interest in the software to be built. In addition, the questions identify the measurable benefit of a successful implementation and possible alternatives to custom software development.

The next set of questions enables you to gain a better understanding of the problem and allows the customer to voice his or her perceptions about a solution:

? What questions will help you gain a preliminary understanding of the problem?

- How would you characterize "good" output that would be generated by a successful solution?
- What problem(s) will this solution address?
- Can you show me (or describe) the business environment in which the solution will be used?
- Will special performance issues or constraints affect the way the solution is approached?

The final set of questions focuses on the effectiveness of the communication activity itself. Gause and Weinberg [Gau89] call these "meta-questions" and propose the following (abbreviated) list:

- Are you the right person to answer these questions? Are your answers "official"?
- Are my questions relevant to the problem that you have?
- Am I asking too many questions?
- Can anyone else provide additional information?
- Should I be asking you anything else?

These questions (and others) will help to "break the ice" and initiate the communication that is essential to successful elicitation. But a question-and-answer meeting format is not an approach that has been overwhelmingly successful. In fact, the Q&A session should be used for the first encounter only and then replaced by a requirements elicitation format that combines elements of problem solving, negotiation, and specification. An approach of this type is presented in Section 5.3.

5.3   ELICITING REQUIREMENTS

Requirements elicitation (also called *requirements gathering*) combines elements of problem solving, elaboration, negotiation, and specification. In order to encourage a collaborative, team-oriented approach to requirements gathering, stakeholders work together to identify the problem, propose elements of the solution, negotiate different approaches and specify a preliminary set of solution requirements [Zah90].[9]

5.3.1   Collaborative Requirements Gathering

Many different approaches to collaborative requirements gathering have been proposed. Each makes use of a slightly different scenario, but all apply some variation on the following basic guidelines:

- Meetings are conducted and attended by both software engineers and other stakeholders.
- Rules for preparation and participation are established.
- An agenda is suggested that is formal enough to cover all important points but informal enough to encourage the free flow of ideas.
- A "facilitator" (can be a customer, a developer, or an outsider) controls the meeting.
- A "definition mechanism" (can be work sheets, flip charts, or wall stickers or an electronic bulletin board, chat room, or virtual forum) is used.

---

9   This approach is sometimes called a *facilitated application specification technique* (FAST).

The goal is to identify the problem, propose elements of the solution, negotiate different approaches, and specify a preliminary set of solution requirements in an atmosphere that is conducive to the accomplishment of the goal. To better understand the flow of events as they occur, I present a brief scenario that outlines the sequence of events that lead up to the requirements gathering meeting, occur during the meeting, and follow the meeting.

During inception (Section 5.2) basic questions and answers establish the scope of the problem and the overall perception of a solution. Out of these initial meetings, the developer and customers write a one- or two-page "product request."

A meeting place, time, and date are selected; a facilitator is chosen; and attendees from the software team and other stakeholder organizations are invited to participate. The product request is distributed to all attendees before the meeting date.

As an example,[10] consider an excerpt from a product request written by a marketing person involved in the *SafeHome* project. This person writes the following narrative about the home security function that is to be part of *SafeHome:*

> Our research indicates that the market for home management systems is growing at a rate of 40 percent per year. The first *SafeHome* function we bring to market should be the home security function. Most people are familiar with "alarm systems" so this would be an easy sell.
>
> The home security function would protect against and/or recognize a variety of undesirable "situations" such as illegal entry, fire, flooding, carbon monoxide levels, and others. It'll use our wireless sensors to detect each situation. It can be programmed by the homeowner, and will automatically telephone a monitoring agency when a situation is detected.

In reality, others would contribute to this narrative during the requirements gathering meeting and considerably more information would be available. But even with additional information, ambiguity would be present, omissions would likely exist, and errors might occur. For now, the preceding "functional description" will suffice.

While reviewing the product request in the days before the meeting, each attendee is asked to make a list of objects that are part of the environment that surrounds the system, other objects that are to be produced by the system, and objects that are used by the system to perform its functions. In addition, each attendee is asked to make another list of services (processes or functions) that manipulate or interact with the objects. Finally, lists of constraints (e.g., cost, size, business rules) and performance criteria (e.g., speed, accuracy) are also developed. The attendees are informed that the lists are not expected to be exhaustive but are expected to reflect each person's perception of the system.

---

10 This example (with extensions and variations) is used to illustrate important software engineering methods in many of the chapters that follow. As an exercise, it would be worthwhile to conduct your own requirements gathering meeting and develop a set of lists for it.

Objects described for *SafeHome* might include the control panel, smoke detectors, window and door sensors, motion detectors, an alarm, an event (a sensor has been activated), a display, a PC, telephone numbers, a telephone call, and so on. The list of services might include *configuring* the system, *setting* the alarm, *monitoring* the sensors, *dialing* the phone, *programming* the control panel, and *reading* the display (note that services act on objects). In a similar fashion, each attendee will develop lists of constraints (e.g., the system must recognize when sensors are not operating, must be user-friendly, must interface directly to a standard phone line) and performance criteria (e.g., a sensor event should be recognized within one second, and an event priority scheme should be implemented).

The lists of objects can be pinned to the walls of the room using large sheets of paper, stuck to the walls using adhesive-backed sheets, or written on a wall board. Alternatively, the lists may have been posted on an electronic bulletin board, at an internal website, or posed in a chat room environment for review prior to the meeting. Ideally, each listed entry should be capable of being manipulated separately so that lists can be combined, entries can be modified, and additions can be made. At this stage, critique and debate are strictly prohibited.

After individual lists are presented in one topic area, the group creates a combined list by eliminating redundant entries, adding any new ideas that come up during the discussion, but not deleting anything. After you create combined lists for all topic areas, discussion—coordinated by the facilitator—ensues. The combined list is shortened, lengthened, or reworded to properly reflect the product/system to be developed. The objective is to develop a consensus list of objects, services, constraints, and performance for the system to be built.

In many cases, an object or service described on a list will require further explanation. To accomplish this, stakeholders develop *mini-specifications* for entries on the lists.[11] Each mini-specification is an elaboration of an object or service. For example, the mini-spec for the *SafeHome* object **Control Panel** might be:

> The control panel is a wall-mounted unit that is approximately 9 × 5 inches in size. The control panel has wireless connectivity to sensors and a PC. User interaction occurs through a keypad containing 12 keys. A 3 × 3 inch LCD color display provides user feedback. Software provides interactive prompts, echo, and similar functions.

The mini-specs are presented to all stakeholders for discussion. Additions, deletions, and further elaboration are made. In some cases, the development of mini-specs will uncover new objects, services, constraints, or performance requirements that will be added to the original lists. During all discussions, the team may raise an issue that cannot be resolved during the meeting. An *issues list* is maintained so that these ideas will be acted on later.

---

11 Rather than creating a mini-specification, many software teams elect to develop user scenarios called *use cases.* These are considered in detail in Section 5.4 and in Chapter 6.

SafeHome

Conducting a Requirements Gathering Meeting

The scene: A meeting room. The first requirements gathering meeting is in progress.

The players: Jamie Lazar, software team member; Vinod Raman, software team member; Ed Robbins, software team member; Doug Miller, software engineering manager; three members of marketing; a product engineering representative; and a facilitator.

The conversation:

Facilitator (pointing at whiteboard): So that's the current list of objects and services for the home security function.

Marketing person: That about covers it from our point of view.

Vinod: Didn't someone mention that they wanted all *SafeHome* functionality to be accessible via the Internet? That would include the home security function, no?

Marketing person: Yes, that's right . . . we'll have to add that functionality and the appropriate objects.

Facilitator: Does that also add some constraints?

Jamie: It does, both technical and legal.

Production rep: Meaning?

Jamie: We better make sure an outsider can't hack into the system, disarm it, and rob the place or worse. Heavy liability on our part.

Doug: Very true.

Marketing: But we still need that . . . just be sure to stop an outsider from getting in.

Ed: That's easier said than done and . . .

Facilitator (interrupting): I don't want to debate this issue now. Let's note it as an action item and proceed.

(Doug, serving as the recorder for the meeting, makes an appropriate note.)

Facilitator: I have a feeling there's still more to consider here.

(The group spends the next 20 minutes refining and expanding the details of the home security function.)

5.3.2   Quality Function Deployment

Quality function deployment (QFD) is a quality management technique that translates the needs of the customer into technical requirements for software. QFD "concentrates on maximizing customer satisfaction from the software engineering process" [Zul92]. To accomplish this, QFD emphasizes an understanding of what is valuable to the customer and then deploys these values throughout the engineering process. QFD identifies three types of requirements [Zul92]:

KEY POINT

QFD defines requirements in a way that maximizes customer satisfaction.

ADVICE

Everyone wants to implement lots of exciting requirements, but be careful. That's how "requirements creep" sets in. On the other hand, exciting requirements lead to a breakthrough product!

Normal requirements.   The objectives and goals that are stated for a product or system during meetings with the customer. If these requirements are present, the customer is satisfied. Examples of normal requirements might be requested types of graphical displays, specific system functions, and defined levels of performance.

Expected requirements.   These requirements are implicit to the product or system and may be so fundamental that the customer does not explicitly state them. Their absence will be a cause for significant dissatisfaction. Examples of expected requirements are: ease of human/machine interaction, overall operational correctness and reliability, and ease of software installation.

Exciting requirements.   These features go beyond the customer's expectations and prove to be very satisfying when present. For example, software for a new mobile phone comes with standard features, but is coupled with a set of unexpected capabilities (e.g., multitouch screen, visual voice mail) that delight every user of the product.

Although QFD concepts can be applied across the entire software process [Par96a], specific QFD techniques are applicable to the requirements elicitation activity. QFD uses customer interviews and observation, surveys, and examination of historical data (e.g., problem reports) as raw data for the requirements gathering activity. These data are then translated into a table of requirements—called the *customer voice table*—that is reviewed with the customer and other stakeholders. A variety of diagrams, matrices, and evaluation methods are then used to extract expected requirements and to attempt to derive exciting requirements [Aka04].

5.3.3   Usage Scenarios

As requirements are gathered, an overall vision of system functions and features begins to materialize. However, it is difficult to move into more technical software engineering activities until you understand how these functions and features will be used by different classes of end users. To accomplish this, developers and users can create a set of scenarios that identify a thread of usage for the system to be constructed. The scenarios, often called *use cases* [Jac92], provide a description of how the system will be used. Use cases are discussed in greater detail in Section 5.4.

SAFEHOME

Developing a Preliminary User Scenario

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

The players: Jamie Lazar, software team member; Vinod Raman, software team member; Ed Robbins, software team member; Doug Miller, software engineering manager; three members of marketing; a product engineering representative; and a facilitator.

The conversation:

Facilitator: We've been talking about security for access to *SafeHome* functionality that will be accessible via the Internet. I'd like to try something. Let's develop a usage scenario for access to the home security function.

Jamie: How?

Facilitator: We can do it a couple of different ways, but for now, I'd like to keep things really informal. Tell us (he points at a marketing person) how you envision accessing the system.

Marketing person: Um . . . well, this is the kind of thing I'd do if I was away from home and I had to let someone into the house, say a housekeeper or repair guy, who didn't have the security code.

Facilitator (smiling): That's the reason you'd do it . . . tell me how you'd actually do this.

Marketing person: Um . . . the first thing I'd need is a PC. I'd log on to a website we'd maintain for all users of *SafeHome*. I'd provide my user id and . . .

Vinod (interrupting): The Web page would have to be secure, encrypted, to guarantee that we're safe and . . .

Facilitator (interrupting): That's good information, Vinod, but it's technical. Let's just focus on how the end user will use this capability. OK?

Vinod: No problem.

Marketing person: So as I was saying, I'd log on to a website and provide my user ID and two levels of passwords.

Jamie: What if I forget my password?

Facilitator (interrupting): Good point, Jamie, but let's not address that now. We'll make a note of that and call it an *exception*. I'm sure there'll be others.

Marketing person: After I enter the passwords, a screen representing all *SafeHome* functions will appear. I'd select the home security function. The system might request that I verify who I am, say, by asking for my address or phone number or something. It would then display a picture of the security system control panel

along with a list of functions that I can perform—arm the system, disarm the system, disarm one or more sensors. I suppose it might also allow me to reconfigure security zones and other things like that, but I'm not sure.

(As the marketing person continues talking, Doug takes copious notes; these form the basis for the first informal usage scenario. Alternatively, the marketing person could have been asked to write the scenario, but this would be done outside the meeting.)

5.3.4 Elicitation Work Products

The work products produced as a consequence of requirements elicitation will vary depending on the size of the system or product to be built. For most systems, the work products include

? **What information is produced as a consequence of requirements gathering?**

- A statement of need and feasibility.
- A bounded statement of scope for the system or product.
- A list of customers, users, and other stakeholders who participated in requirements elicitation.
- A description of the system's technical environment.
- A list of requirements (preferably organized by function) and the domain constraints that apply to each.
- A set of usage scenarios that provide insight into the use of the system or product under different operating conditions.
- Any prototypes developed to better define requirements.

Each of these work products is reviewed by all people who have participated in requirements elicitation.

5.4 DEVELOPING USE CASES

In a book that discusses how to write effective use cases, Alistair Cockburn [Coc01b] notes that "a use case captures a contract … [that] describes the system's behavior under various conditions as the system responds to a request from one of its stakeholders . . ." In essence, a use case tells a stylized story about how an end user (playing one of a number of possible roles) interacts with the system under a specific set of circumstances. The story may be narrative text, an outline of tasks or interactions, a template-based description, or a diagrammatic representation. Regardless of its form, a use case depicts the software or system from the end user's point of view.

The first step in writing a use case is to define the set of "actors" that will be involved in the story. *Actors* are the different people (or devices) that use the system or product within the context of the function and behavior that is to be described. Actors represent the roles that people (or devices) play as the system operates. Defined somewhat more formally, an actor is anything that communicates with the system or product and that is external to the system itself. Every actor has one or more goals when using the system.

It is important to note that an actor and an end user are not necessarily the same thing. A typical user may play a number of different roles when using a system, whereas an actor represents a class of external entities (often, but not always, people) that play just one role in the context of the use case. As an example, consider a machine operator (a user) who interacts with the control computer for a manufacturing cell that contains a number of robots and numerically controlled machines. After careful review of requirements, the software for the control computer requires four different modes (roles) for interaction: programming mode, test mode, monitoring mode, and troubleshooting mode. Therefore, four actors can be defined: programmer, tester, monitor, and troubleshooter. In some cases, the machine operator can play all of these roles. In others, different people may play the role of each actor.

Because requirements elicitation is an evolutionary activity, not all actors are identified during the first iteration. It is possible to identify primary actors [Jac92] during the first iteration and secondary actors as more is learned about the system. *Primary actors* interact to achieve required system function and derive the intended benefit from the system. They work directly and frequently with the software. *Secondary actors* support the system so that primary actors can do their work.

Once actors have been identified, use cases can be developed. Jacobson [Jac92] suggests a number of questions[12] that should be answered by a use case:

? What do I need to know in order to develop an effective use case?

- Who is the primary actor, the secondary actor(s)?
- What are the actor's goals?
- What preconditions should exist before the story begins?
- What main tasks or functions are performed by the actor?
- What exceptions might be considered as the story is described?
- What variations in the actor's interaction are possible?
- What system information will the actor acquire, produce, or change?
- Will the actor have to inform the system about changes in the external environment?
- What information does the actor desire from the system?
- Does the actor wish to be informed about unexpected changes?

---

12 Jacobson's questions have been extended to provide a more complete view of use-case content.
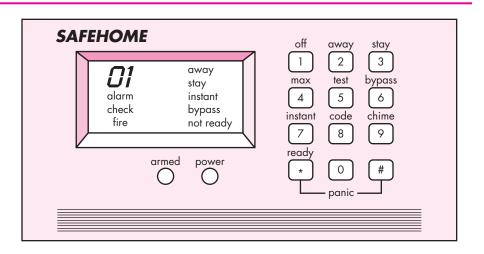
Recalling basic *SafeHome* requirements, we define four actors: **homeowner** (a user), **setup manager** (likely the same person as **homeowner,** but playing a different role), **sensors** (devices attached to the system), and the **monitoring and response subsystem** (the central station that monitors the *SafeHome* home security function). For the purposes of this example, we consider only the **homeowner** actor. The **homeowner** actor interacts with the home security function in a number of different ways using either the alarm control panel or a PC:

- Enters a password to allow all other interactions.
- Inquires about the status of a security zone.
- Inquires about the status of a sensor.
- Presses the panic button in an emergency.
- Activates/deactivates the security system.

Considering the situation in which the homeowner uses the control panel, the basic use case for system activation follows:[13]

1. The homeowner observes the *SafeHome* control panel (Figure 5.1) to determine if the system is ready for input. If the system is not ready, a *not ready* message is displayed on the LCD display, and the homeowner must physically close windows or doors so that the *not ready* message disappears. [A *not ready* message implies that a sensor is open; i.e., that a door or window is open.]

FIGURE 5.1

SafeHome control panel

---

13 Note that this use case differs from the situation in which the system is accessed via the Internet. In this case, interaction occurs via the control panel, not the graphical user interface (GUI) provided when a PC is used.

2. 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.

3. The homeowner selects and keys in *stay* or *away* (see Figure 5.1) to activate the system. *Stay* activates only perimeter sensors (inside motion detecting sensors are deactivated). *Away* activates all sensors.

4. When activation occurs, a red alarm light can be observed by the homeowner.

The basic use case presents a high-level story that describes the interaction between the actor and the system.

In many instances, uses cases are further elaborated to provide considerably more detail about the interaction. For example, Cockburn [Coc01b] suggests the following template for detailed descriptions of use cases:

Use case:	*InitiateMonitoring*
Primary actor:	Homeowner.
Goal in context:	To set the system to monitor sensors when the homeowner leaves the house or remains inside.
Preconditions:	System has been programmed for a password and to recognize various sensors.
Trigger:	The homeowner decides to "set" the system, i.e., to turn on the alarm functions.

Scenario:

1. Homeowner: observes control panel

2. Homeowner: enters password

3. Homeowner: selects "stay" or "away"

4. Homeowner: observes read alarm light to indicate that *SafeHome* has been armed

Exceptions:

1. Control panel is *not ready:* homeowner checks all sensors to determine which are open; closes them.

2. Password is incorrect (control panel beeps once): homeowner reenters correct password.

3. Password not recognized: monitoring and response subsystem must be contacted to reprogram password.

4. *Stay* is selected: control panel beeps twice and a *stay* light is lit; perimeter sensors are activated.

5. *Away* is selected: control panel beeps three times and an *away* light is lit; all sensors are activated.

Priority:	Essential, must be implemented
When available:	First increment

Frequency of use:	Many times per day
Channel to actor:	Via control panel interface
Secondary actors:	Support technician, sensors

Channels to secondary actors:

Support technician: phone line

Sensors: hardwired and radio frequency interfaces

Open issues:

1. Should there be a way to activate the system without the use of a password or with an abbreviated password?

2. Should the control panel display additional text messages?

3. How much time does the homeowner have to enter the password from the time the first key is pressed?

4. Is there a way to deactivate the system before it actually activates?

Use cases for other **homeowner** interactions would be developed in a similar manner. It is important to review each use case with care. If some element of the interaction is ambiguous, it is likely that a review of the use case will indicate a problem.

SAFEHOME

Developing a High-Level Use-Case Diagram

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

The players: Jamie Lazar, software team member; Vinod Raman, software team member; Ed Robbins, software team member; Doug Miller, software engineering manager; three members of marketing; a product engineering representative; and a facilitator.

The conversation:

Facilitator: We've spent a fair amount of time talking about *SafeHome* home security functionality. During the break I sketched a use case diagram to summarize the important scenarios that are part of this function. Take a look.

(All attendees look at Figure 5.2.)

Jamie: I'm just beginning to learn UML notation.[14] So the home security function is represented by the big box with the ovals inside it? And the ovals represent use cases that we've written in text?

Facilitator: Yep. And the stick figures represent actors—the people or things that interact with the system as described by the use case . . . oh, I use the labeled square to represent an actor that's not a person . . . in this case, sensors.

Doug: Is that legal in UML?

Facilitator: Legality isn't the issue. The point is to communicate information. I view the use of a humanlike stick figure for representing a device to be misleading. So I've adapted things a bit. I don't think it creates a problem.

Vinod: Okay, so we have use-case narratives for each of the ovals. Do we need to develop the more detailed template-based narratives I've read about?

Facilitator: Probably, but that can wait until we've considered other *SafeHome* functions.
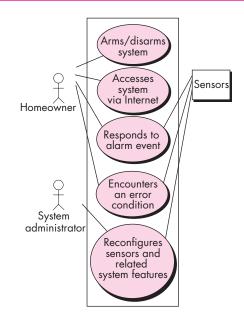
Marketing person: Wait, I've been looking at this diagram and all of a sudden I realize we missed something.

Facilitator: Oh really. Tell me what we've missed.

(The meeting continues.)

---

14 A brief UML tutorial is presented in Appendix 1 for those who are unfamiliar with the notation.

FIGURE 5.2

UML use case diagram for *SafeHome* home security function

SOFTWARE TOOLS

Use-Case Development

Objective: Assist in the development of use cases by providing automated templates and mechanisms for assessing clarity and consistency.

Mechanics: Tool mechanics vary. In general, use-case tools provide fill-in-the-blank templates for creating effective use cases. Most use-case functionality is embedded into a set of broader requirements engineering functions.

Representative Tools:[15]

The vast majority of UML-based analysis modeling tools provide both text and graphical support for use-case development and modeling.

Objects by Design
(**www.objectsbydesign.com/tools/ umltools_byCompany.html**) provides comprehensive links to tools of this type.

5.5   BUILDING THE REQUIREMENTS MODEL[16]

The intent of the analysis model is to provide a description of the required informational, functional, and behavioral domains for a computer-based system. The model changes dynamically as you learn more about the system to be built, and other stakeholders understand more about what they really require. For that reason, the analysis model is a snapshot of requirements at any given time. You should expect it to change.

---

15 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.

16 Throughout this book, I use the terms *analysis model* and *requirements model* synonymously. Both refer to representations of the information, functional, and behavioral domains that describe problem requirements.

As the requirements model evolves, certain elements will become relatively stable, providing a solid foundation for the design tasks that follow. However, other elements of the model may be more volatile, indicating that stakeholders do not yet fully understand requirements for the system. The analysis model and the methods that are used to build it are presented in detail in Chapters 6 and 7. I present a brief overview in the sections that follow.

5.5.1 Elements of the Requirements Model

There are many different ways to look at the requirements for a computer-based system. Some software people argue that it's best to select one mode of representation (e.g., the use case) and apply it to the exclusion of all other modes. Other practitioners believe that it's worthwhile to use a number of different modes of representation to depict the requirements model. Different modes of representation force you to consider requirements from different viewpoints—an approach that has a higher probability of uncovering omissions, inconsistencies, and ambiguity.

The specific elements of the requirements model are dictated by the analysis modeling method (Chapters 6 and 7) that is to be used. However, a set of generic elements is common to most requirements models.

Scenario-based elements. The system is described from the user's point of view using a scenario-based approach. For example, basic use cases (Section 5.4) and their corresponding use-case diagrams (Figure 5.2) evolve into more elaborate template-based use cases. Scenario-based elements of the requirements model are often the first part of the model that is developed. As such, they serve as input for the creation of other modeling elements. Figure 5.3 depicts a UML activity diagram[17] for eliciting requirements and representing them using use cases. Three levels of elaboration are shown, culminating in a scenario-based representation.

Class-based elements. Each usage scenario implies a set of objects that are manipulated as an actor interacts with the system. These objects are categorized into classes—a collection of things that have similar attributes and common behaviors. For example, a UML class diagram can be used to depict a **Sensor** class for the *SafeHome* security function (Figure 5.4). Note that the diagram lists the attributes of sensors (e.g., name, type) and the operations (e.g., *identify, enable*) that can be applied to modify these attributes. In addition to class diagrams, other analysis modeling elements depict the manner in which classes collaborate with one another and the relationships and interactions between classes. These are discussed in more detail in Chapter 7.

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

17 A brief UML tutorial is presented in Appendix 1 for those who are unfamiliar with the notation.

FIGURE 5.3

**UML activity
diagrams for
eliciting
requirements**
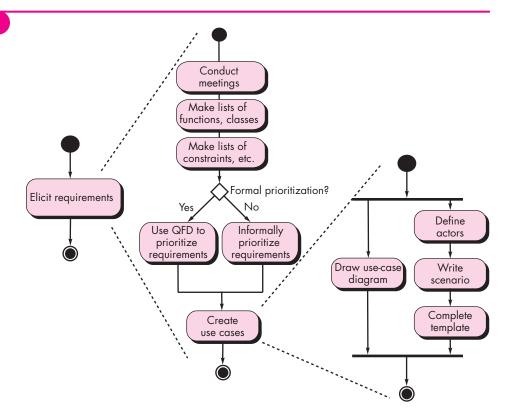
FIGURE 5.4

**Class diagram
for sensor**

KEY
POINT
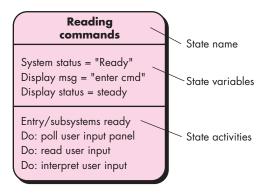
A state is an externally
observable mode of
behavior. External
stimuli cause transi-
tions between states.

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

To illustrate the use of a state diagram, consider software embedded within the
SafeHome control panel that is responsible for reading user input. A simplified UML
state diagram is shown in Figure 5.5.

UML state
diagram
notation


```
Reading
commands

System status = "Ready"          State name
Display msg = "enter cmd"
Display status = steady          State variables

Entry/subsystems ready
Do: poll user input panel
Do: read user input              State activities
Do: interpret user input
```

In addition to behavioral representations of the system as a whole, the behavior of individual classes can also be modeled. Further discussion of behavioral modeling is presented in Chapter 7.

SafeHome

Preliminary Behavioral Modeling

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

The players: Jamie Lazar, software team member; Vinod Raman, software team member; Ed Robbins, software team member; Doug Miller, software engineering manager; three members of marketing; a product engineering representative; and a facilitator.

The conversation:

Facilitator: We've just about finished talking about *SafeHome* home security functionality. But before we do, I want to discuss the behavior of the function.

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

Ed (smiling): That's when you give the product a "timeout" if it misbehaves.

Facilitator: Not exactly. Let me explain.

(The facilitator explains the basics of behavioral modeling to the requirements gathering team.)

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

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

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

Facilitator: See, you can do it.

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

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

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

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

Flow-oriented elements. Information is transformed as it flows through a computer-based system. The system accepts input in a variety of forms, applies functions to transform it, and produces output in a variety of forms. Input may be a control signal transmitted by a transducer, a series of numbers typed by a human operator, a

packet of information transmitted on a network link, or a voluminous data file retrieved from secondary storage. The transform(s) may comprise a single logical comparison, a complex numerical algorithm, or a rule-inference approach of an expert system. Output may light a single LED or produce a 200-page report. In effect, we can create a flow model for any computer-based system, regardless of size and complexity. A more detailed discussion of flow modeling is presented in Chapter 7.

5.5.2   Analysis Patterns

Anyone who has done requirements engineering on more than a few software projects begins to notice that certain problems reoccur across all projects within a specific application domain.[18] These *analysis patterns* [Fow97] suggest solutions (e.g., a class, a function, a behavior) within the application domain that can be reused when modeling many applications.

Geyer-Schulz and Hahsler [Gey01] suggest two benefits that can be associated with the use of analysis patterns:

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

Analysis patterns are integrated into the analysis model by reference to the pattern name. They are also stored in a repository so that requirements engineers can use search facilities to find and apply them. Information about an analysis pattern (and other types of patterns) is presented in a standard template [Gey01][19] that is discussed in more detail in Chapter 12. Examples of analysis patterns and further discussion of this topic are presented in Chapter 7.

5.6   NEGOTIATING REQUIREMENTS

In an ideal requirements engineering context, the inception, elicitation, and elaboration tasks determine customer requirements in sufficient detail to proceed to subsequent software engineering activities. Unfortunately, this rarely happens. In reality, you may have to enter into a negotiation with one or more stakeholders. In most cases, stakeholders are 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 stakeholder needs while at the

---

18 In some cases, problems reoccur regardless of the application domain. For example, the features and functions used to solve user interface problems are common regardless of the application domain under consideration.

19 A variety of patterns templates have been proposed in the literature. If you have interest, see [Fow97], [Gam95], [Yac03], and [Bus07] among many sources.

same time reflecting the real-world constraints (e.g., time, people, budget) that have been placed on the software team.

The best negotiations strive for a "win-win" result.[20] That is, stakeholders win by getting the system or product that satisfies the majority of their needs and you (as a member of the software team) win by working to realistic and achievable budgets and deadlines.

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. Negotiation 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

to her. 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.

The conversation:

Lisa: So, I hear the first meeting went really well.

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

---

20 Dozens of books have been written on negotiating skills (e.g., [Lew06], [Rai06], [Fis06]). It is one of the more important skills that you can learn. Read one.

Lisa (smiling): 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 beanie the next time I visit . . . Look, Lisa, I think we may have a problem with getting all of the functionality for the home security system 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 (frowning): 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 'til the second release.

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

Doug: I understand your situation, I really do. The problem is that in order to give you Internet access, we'll have to have a fully secure website up and running. That takes time and people. We'll also have to build a lot of additional functionality into the first release . . . I don't think we can do it with the resources we've got.

Lisa (still frowning): I see, but you've got to figure out a way to get it done. It's pivotal to home security functions and to other functions as well . . . those can wait until the next releases . . . I'll agree to that.

Lisa and Doug appear to be at an impasse, and yet they must negotiate a solution to this problem. Can they both "win" here? Playing the role of a mediator, what would you suggest?

5.7   VALIDATING REQUIREMENTS

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

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

- Is each requirement consistent with the overall objectives 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?

These and other questions should be asked and answered to ensure that the requirements model is an accurate reflection of stakeholder needs and that it provides a solid foundation for design.

5.8 Summary

Requirements engineering tasks are conducted to establish a solid foundation for design and construction. Requirements engineering occurs during the communication and modeling activities that have been defined for the generic software process. Seven distinct requirements engineering functions—inception, elicitation, elaboration, negotiation, specification, validation, and management—are conducted by members of the software team.

At project inception, stakeholders establish basic problem requirements, define overriding project constraints, and address major features and functions that must be present for the system to meet its objectives. This information is refined and expanded during elicitation—a requirements gathering activity that makes use of facilitated meetings, QFD, and the development of usage scenarios.

Elaboration further expands requirements in a model—a collection of scenario-based, class-based, behavioral, and flow-oriented elements. The model may reference analysis patterns, solutions for analysis problems that have been seen to reoccur across different applications.

As requirements are identified and the requirements model is being created, the software team and other project stakeholders negotiate the priority, availability, and relative cost of each requirement. The intent of this negotiation is to develop a realistic project plan. In addition, each requirement and the requirements model as a whole are validated against customer need to ensure that the right system is to be built.

Problems and Points to Ponder

5.1. Why is it that many software developers don't pay enough attention to requirements engineering? Are there ever circumstances where you can skip it?

5.2. You have been given the responsibility to elicit requirements from a customer who tells you he is too busy to meet with you. What should you do?

5.3. Discuss some of the problems that occur when requirements must be elicited from three or four different customers.

5.4. Why do we say that the requirements model represents a snapshot of a system in time?

5.5. Let's assume that you've convinced the customer (you're a very good salesperson) to agree to every demand that you have as a developer. Does that make you a master negotiator? Why?

5.6. Develop at least three additional "context-free questions" that you might ask a stakeholder during inception.

5.7. Develop a requirements gathering "kit." The kit should include a set of guidelines for conducting a requirements gathering meeting and materials that can be used to facilitate the creation of lists and any other items that might help in defining requirements.

5.8. Your instructor will divide the class into groups of four to six students. Half of the group will play the role of the marketing department and half will take on the role of software engineering. Your job is to define requirements for the *SafeHome* security function described in this chapter. Conduct a requirements gathering meeting using the guidelines presented in this chapter.

5.9. Develop a complete use case for one of the following activities:

   a. Making a withdrawal at an ATM
   b. Using your charge card for a meal at a restaurant
   c. Buying a stock using an on-line brokerage account
   d. Searching for books (on a specific topic) using an on-line bookstore
   e. An activity specified by your instructor.

5.10. What do use case "exceptions" represent?

5.11. Describe what an *analysis pattern* is in your own words.

5.12. Using the template presented in Section 5.5.2, suggest one or more analysis pattern for the following application domains:

   a. Accounting software
   b. E-mail software
   c. Internet browsers
   d. Word-processing software
   e. Website creation software
   f. An application domain specified by your instructor.

5.13. What does *win-win* mean in the context of negotiation during the requirements engineering activity?

5.14. What do you think happens when requirement validation uncovers an error? Who is involved in correcting the error?

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. Hood and his colleagues (*Requirements Management,* Springer, 2007) discuss a variety of requirements engineering issues that span both systems and software engineering. Young (*The Requirements Engineering Handbook,* Artech House Publishers, 2007) presents an in-depth discussion of requirements engineering tasks. Wiegers (*More About Software Requirements,* Microsoft Press, 2006) provides many practical techniques for requirements gathering and management. Hull and her colleagues (*Requirements Engineering,* 2d ed., Springer-Verlag, 2004), 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. Gottesdiener (*Requirements by Collaboration: Workshops for Defining*

Needs, Addison-Wesley, 2002) provides useful guidance for those who must establish a collaborative requirements gathering environment with stakeholders.

Lauesen (*Software Requirements: Styles and Techniques,* Addison-Wesley, 2002) presents a comprehensive survey of requirement analysis methods and notation. Weigers (*Software Requirements,* Microsoft Press, 1999) and Leffingwell and his colleagues (*Managing Software Requirements: A Use Case Approach,* 2d ed., Addison-Wesley, 2003) present a useful collection of requirement best practices and suggest pragmatic guidelines for most aspects of the requirements engineering process.

A patterns-based view of requirements engineering is described by Withall (*Software Requirement Patterns,* Microsoft Press, 2007). 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 Rosenberg and Stephens (*Use Case Driven Object Modeling with UML: Theory and Practice,* Apress, 2007), Denny (*Succeeding with Use Cases: Working Smart to Deliver Quality,* Addison-Wesley, 2005), Alexander and Maiden (eds.) (*Scenarios, Stories, Use Cases: Through the Systems Development Life-Cycle,* Wiley, 2004), Leffingwell and his colleagues (*Managing Software Requirements: A Use Case Approach,* 2d ed., Addison-Wesley, 2003) present a useful collection of requirement best practices. Bittner and Spence (*Use Case Modeling,* Addison-Wesley, 2002), Cockburn [Coc01], Armour and Miller (*Advanced Use Case Modeling: Software Systems,* Addison-Wesley, 2000), and Kulak and his colleagues (*Use Cases: Requirements in Context,* Addison-Wesley, 2000) discuss requirements gathering with an emphasis on use-case modeling.

A wide variety of information sources on requirements engineering and analysis is 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 website: **www.mhhe.com/engcs/ compsci/pressman/professional/olc/ser.htm**.

REQUIREMENTS MODELING: SCENARIOS, INFORMATION, AND ANALYSIS CLASSES

At a technical level, software engineering begins with a series of modeling tasks that lead to a specification of requirements and a design representation for the software to be built. The requirements model[1]— actually a set of models—is the first technical representation of a system.

In a seminal book on requirements modeling methods, Tom DeMarco [DeM79] describes the process in this way:

> Looking back over the recognized problems and failings of the analysis phase, I suggest that we need to make the following additions to our set of analysis phase goals. The products of analysis must be highly maintainable. This applies particularly to the

QUICK LOOK

What is it? The written word is a wonderful vehicle for communication, but it is not necessarily the best way to represent the requirements for computer software. Requirements modeling uses a combination of text and diagrammatic forms to depict requirements in a way that is relatively easy to understand, and more important, straightforward to review for correctness, completeness, and consistency.

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

Why is it important? To validate software requirements, you need to examine them from a number of different points of view. In this chapter you'll consider requirements modeling from three different perspectives: scenario-based models, data (information) models, and class-based models. Each represents requirements in a different "dimension," thereby increasing the probability that errors will be found, that inconsistency will surface, and that omissions will be uncovered.

What are the steps? Scenario-based modeling represents the system from the user's point of view. Data modeling represents the information space and depicts the data objects that the software will manipulate and the relationships among them. Class-based modeling defines objects, attributes, and relationships. Once preliminary models are created, they are refined and analyzed to assess their clarity, completeness, and consistency. In Chapter 7, we extend the modeling dimensions noted here with additional representations, providing a more robust view of requirements.

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

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

---

1   In past editions of this book, I used the term *analysis model,* rather than *requirements model.* In this edition, I've decided to use both phrases to represent the modeling activity that defines various aspects of the problem to be solved. *Analysis* is the action that occurs as *requirements* are derived.

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 century ago, his comments still apply to modern requirements modeling methods and notation.

6.1 REQUIREMENTS ANALYSIS

Requirements analysis results in the specification of software's operational characteristics, indicates software's interface with other system elements, and establishes constraints that software must meet. Requirements analysis allows you (regardless of whether you're called a *software engineer,* an *analyst,* or a *modeler*) to elaborate on basic requirements established during the inception, elicitation, and negotiation tasks that are part of requirements engineering (Chapter 5).

The requirements modeling action results in one or more of the following types of models:

- *Scenario-based models* of requirements from the point of view of various system "actors"
- *Data models* that depict the information domain for the problem
- *Class-oriented models* that represent object-oriented classes (attributes and operations) and the manner in which classes collaborate to achieve system requirements
- *Flow-oriented models* that represent the functional elements of the system and how they transform data as it moves through the system
- *Behavioral models* that depict how the software behaves as a consequence of external "events"

> uote:
>
> "Any one 'view' of requirements is insufficient to understand or describe the desired behavior of a complex system."
>
> **Alan M. Davis**

KEY POINT
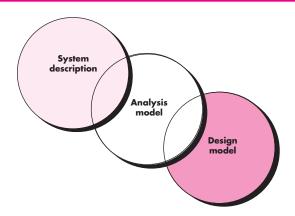
The analysis model and requirements specification provide a means for assessing quality once the software is built.

These models provide a software designer with information that can be translated to architectural, interface, and component-level designs. Finally, the requirements model (and the software requirements specification) provides the developer and the customer with the means to assess quality once software is built.

In this chapter, I focus on *scenario-based modeling*—a technique that is growing increasingly popular throughout the software engineering community; *data modeling*—a more specialized technique that is particularly appropriate when an application must create or manipulate a complex information space; and *class*

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

modeling—a representation of the object-oriented classes and the resultant collaborations that allow a system to function. Flow-oriented models, behavioral models, pattern-based modeling, and WebApp models are discussed in Chapter 7.

6.1.1 Overall Objectives and Philosophy

Throughout requirements modeling, your primary focus is on *what,* not *how.* What user interaction occurs in a particular circumstance, what objects does the system manipulate, what functions must the system perform, what behaviors does the system exhibit, what interfaces are defined, and what constraints apply?[2]

In earlier chapters, I noted that complete specification of requirements may not be possible at this stage. The customer may be unsure of precisely what is required for certain aspects of the system. The developer may be unsure that a specific approach will properly accomplish function and performance. These realities mitigate in favor of an iterative approach to requirements analysis and modeling. The analyst should model what is known and use that model as the basis for design of the software increment.[3]

The requirements model must achieve three primary objectives: (1) to describe what the customer requires, (2) to establish a basis for the creation of a software design, and (3) to define a set of requirements that can be validated once the software is built. The analysis model bridges the gap between a system-level description that describes overall system or business functionality as it is achieved by applying software, hardware, data, human, and other system elements and a software design (Chapters 8 through 13) that describes the software's application architecture, user interface, and component-level structure. This relationship is illustrated in Figure 6.1.

uote:

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

Andrew Hunt and David Thomas

KEY POINT

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

---

2   It should be noted that as customers become more technologically sophisticated, there is a trend toward the specification of *how* as well as *what.* However, the primary focus should remain on *what.*

3   Alternatively, the software team may choose to create a prototype (Chapter 2) in an effort to better understand requirements for the system.

It is important to note that all elements of the requirements model will be directly traceable to parts of the design model. A clear division of analysis and design tasks between these two important modeling activities is not always possible. Some design invariably occurs as part of analysis, and some analysis will be conducted during design.

6.1.2   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 requirements model should add to an overall understanding of software requirements and provide insight into the information domain, function, and behavior of the system.*

- *Delay consideration of infrastructure and other nonfunctional models until design.* That is, a database may be required, but the classes necessary to implement it, the functions required to access it, and the behavior that will be exhibited as it is used should be considered only after problem domain analysis has been completed.

- *Minimize coupling throughout the system.* It is important to represent relationships between classes and functions. However, if the level of "interconnectedness" is extremely high, effort should be made to reduce it.

- *Be certain that the requirements model provides value to all stakeholders.* Each constituency has its own use for the model. For example, business stakeholders should use the model to validate requirements; designers should use the model as a basis for design; QA people should use the model to help plan acceptance tests.

- *Keep the model as simple as it can be.* Don't create additional diagrams when they add no new information. Don't use complex notational forms, when a simple list will do.

6.1.3   Domain Analysis

In the discussion of requirements engineering (Chapter 5), I noted that analysis patterns often reoccur across many applications within a specific business domain. If these patterns are defined and categorized in a manner that allows you to recognize and apply them to solve common problems, the creation of the analysis model is expedited. More important, the likelihood of applying design patterns and executable software components grows dramatically. This improves time-to-market and reduces development costs.

? **Are there basic guidelines that can help us as we do requirements analysis work?**

Quote:
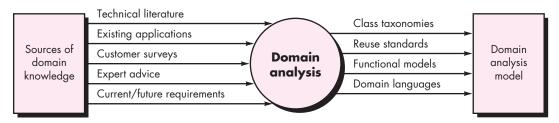
"Problems worthy of attack, prove their worth by hitting back."

Piet Hein

WebRef

Many useful resources for domain analysis can be found at **www.iturls .com/English/ Software Engineering/ SE_mod5.asp**.

FIGURE 6.2   **Input and output for domain analysis**

But how are analysis patterns and classes recognized in the first place? Who defines them, categorizes them, and readies them for use on subsequent projects? The answers to these questions lie in *domain analysis.* Firesmith [Fir93] describes domain analysis in the following way:

> Software domain analysis is the identification, analysis, and specification of common requirements from a specific application domain, typically for reuse on multiple projects within that application domain. . . . [Object-oriented domain analysis is] the identification, analysis, and specification of common, reusable capabilities within a specific application domain, in terms of common objects, classes, subassemblies, and frameworks.

The "specific application domain" can range from avionics to banking, from multimedia video games to software embedded within medical devices. The goal of domain analysis is straightforward: to find or create those analysis classes and/or analysis patterns that are broadly applicable so that they may be reused.[4]

Using terminology that was introduced earlier in this book, domain analysis may be viewed as an umbrella activity for the software process. By this I mean that domain analysis is an ongoing software engineering activity that is not connected to any one software project. In a way, the role of a domain analyst is similar to the role of a master toolsmith in a heavy manufacturing environment. The job of the toolsmith is to design and build tools that may be used by many people doing similar but not necessarily the same jobs. The role of the domain analyst[5] is to discover and define analysis patterns, analysis classes, and related information that may be used by many people working on similar but not necessarily the same applications.

Figure 6.2 [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.

Domain analysis doesn't look at a specific application, but rather at the domain in which the application resides. The intent is to identify common problem solving elements that are applicable to all applications within the domain.

---

4   A complementary view of domain analysis "involves modeling the domain so that software engineers and other stakeholders can better learn about it . . . not all domain classes necessarily result in the development of reusable classes . . ." [Let03a].
5   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.

SAFEHOME

Domain Analysis

The scene: Doug Miller's office, after a meeting with marketing.

The players: Doug Miller, software engineering manager, and Vinod Raman, a member of the software engineering team.

The conversation:

Doug: I need you for a special project, Vinod. I'm going to pull you out of the requirements gathering meetings.

Vinod (frowning): Too bad. That format actually works . . . I was getting something out of it. What's up?

Doug: Jamie and Ed will cover for you. Anyway, marketing insists that we deliver the Internet capability along with the home security function in the first release of *SafeHome*. We're under the gun on this . . . not enough time or people, so we've got to solve both problems—the PC interface and the Web interface—at once.

Vinod (looking confused): I didn't know the plan was set . . . we're not even finished with requirements gathering.

Doug (a wan smile): I know, but the time lines are so short that I decided to begin strategizing with marketing right now . . . anyhow, we'll revisit any tentative plan once we have the info from all of the requirements gathering meetings.

Vinod: Okay, what's up? What do you want me to do?

Doug: Do you know what "domain analysis" is?

Vinod: Sort of. You look for similar patterns in Apps that do the same kinds of things as the App you're building. If possible, you then steal the patterns and reuse them in your work.

Doug: Not sure I like the word *steal,* but basically you have it right. What I'd like you to do is to begin researching existing user interfaces for systems that control something like *SafeHome*. I want you to propose a set of patterns and analysis classes that can be common to both the PC-based interface that'll sit in the house and the browser-based interface that is accessible via the Internet.

Vinod: We can save time by making them the same . . . why don't we just do that?

Doug: Ah . . . it's nice to have people who think like you do. That's the whole point—we can save time and effort if both interfaces are nearly identical, implemented with the same code, blah, blah, that marketing insists on.

Vinod: So you want, what—classes, analysis patterns, design patterns?

Doug: All of 'em. Nothing formal at this point. I just want to get a head start on our internal analysis and design work.

Vinod: I'll go to our class library and see what we've got. I'll also use a patterns template I saw in a book I was reading a few months back.

Doug: Good. Go to work.

6.1.4 Requirements Modeling Approaches

One view of requirements modeling, called *structured analysis,* considers data and the processes that transform the data as separate entities. Data objects are modeled in a way that defines their attributes and relationships. Processes that manipulate data objects are modeled in a manner that shows how they transform data as data objects flow through the system.
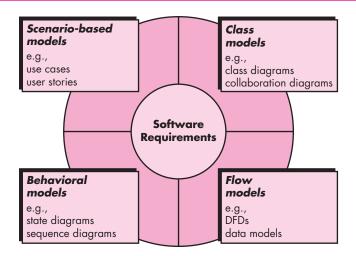
A second approach to analysis modeling, called *object-oriented analysis,* focuses on the definition of classes and the manner in which they collaborate with one another to effect customer requirements. UML and the Unified Process (Chapter 2) are predominantly object oriented.

Although the requirements model proposed in this book combines features of both approaches, software teams often choose one approach and exclude all representations from the other. The question is not which is best, but rather, what

FIGURE 6.3

Elements of the analysis model

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

Scenario-based models
e.g.,
use cases
user stories

Class models
e.g.,
class diagrams
collaboration diagrams

Software Requirements

Behavioral models
e.g.,
state diagrams
sequence diagrams

Flow models
e.g.,
DFDs
data models

uote:

"Why should we build models? Why not just build the system itself? The answer is that we can construct models in such a way as to highlight, or emphasize, certain critical features of a system, while simultaneously de-emphasizing other aspects of the system."

Ed Yourdon

combination of representations will provide stakeholders with the best model of software requirements and the most effective bridge to software design.

Each element of the requirements model (Figure 6.3) presents the problem from a different point of view. Scenario-based elements depict how the user interacts with the system and the specific sequence of activities that occur as the software is used. Class-based elements model the objects that the system will manipulate, the operations that will be applied to the objects to effect the manipulation, relationships (some hierarchical) between the objects, and the collaborations that occur between the classes that are defined. Behavioral elements depict how external events change the state of the system or the classes that reside within it. Finally, flow-oriented elements represent the system as an information transform, depicting how data objects are transformed as they flow through various system functions.

Analysis modeling leads to the derivation of each of these modeling elements. However, the specific content of each element (i.e., the diagrams that are used to construct the element and the model) may differ from project to project. As we have noted a number of times in this book, the software team must work to keep it simple. Only those modeling elements that add value to the model should be used.

6.2   SCENARIO-BASED MODELING

Although the success of a computer-based system or product is measured in many ways, user satisfaction resides at the top of the list. If you understand how end users (and other actors) want to interact with a system, your software team will be better able to properly characterize requirements and build meaningful analysis and design

models. Hence, requirements modeling with UML[6] begins with the creation of scenarios in the form of use cases, activity diagrams, and swimlane diagrams.

6.2.1 Creating a Preliminary Use Case

Alistair Cockburn characterizes a use case as a "contract for behavior" [Coc01b]. As we discussed in Chapter 5, the "contract" defines the way in which an actor[7] uses a computer-based system to accomplish some goal. In essence, a use case captures the interactions that occur between producers and consumers of information and the system itself. In this section, I examine how use cases are developed as part of the requirements modeling activity.[8]

In Chapter 5, I noted that a use case describes a specific usage scenario in straightforward language from the point of view of a defined actor. But how do you know (1) what to write about, (2) how much to write about it, (3) how detailed to make your description, and (4) how to organize the description? These are the questions that must be answered if use cases are to provide value as a requirements modeling tool.

ADVICE

In some situations, use cases become the dominant requirements engineering mechanism. However, this does not mean that you should discard other modeling methods when they are appropriate.

What to write about?   The first two requirements engineering tasks—inception and elicitation—provide you with the information you'll need to begin writing use cases. Requirements gathering meetings, QFD, and other requirements engineering mechanisms are used to identify stakeholders, define the scope of the problem, specify overall operational goals, establish priorities, outline all known functional requirements, and describe the things (objects) that will be manipulated by the system.

To begin developing a set of use cases, list the functions or activities performed by a specific actor. You can obtain these from a list of required system functions, through conversations with stakeholders, or by an evaluation of activity diagrams (Section 6.3.1) developed as part of requirements modeling.

SafeHome

Developing Another Preliminary User Scenario

The scene: A meeting room, during the second requirements gathering meeting.

The players: Jamie Lazar, software team member; Ed Robbins, software team member; Doug Miller, software engineering manager; three members of marketing; a product engineering representative; and a facilitator.

The conversation:

Facilitator: It's time that we begin talking about the *SafeHome* surveillance function. Let's develop a user scenario for access to the surveillance function.

Jamie: Who plays the role of the actor on this?

---

6   UML will be used as the modeling notation throughout this book. Appendix 1 provides a brief tutorial for those readers who may be unfamiliar with basic UML notation.

7   An actor is not a specific person, but rather a role that a person (or a device) plays within a specific context. An actor "calls on the system to deliver one of its services" [Coc01b].

8   Use cases are a particularly important part of analysis modeling for user interfaces. Interface analysis is discussed in detail in Chapter 11.

Facilitator: I think Meredith (a marketing person) has been working on that functionality. Why don't you play the role?

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

Facilitator: Right . . . same way.

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

Ed: Will we use compression to store the video?

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

Meredith: Okay, so basically there are two parts to the surveillance function . . . the first configures the system including laying out a floor plan—we have to have tools to help the homeowner do this—and the second part is the actual surveillance function itself. Since the layout is part of the configuration activity, I'll focus on the surveillance function.

Facilitator (smiling): Took the words right out of my mouth.

Meredith: Um . . . I want to gain access to the surveillance function either via the PC or via the Internet. My feeling is that the Internet access would be more frequently used. Anyway, I want to be able to display camera views on a PC and control pan and zoom for a specific camera. I specify the camera by selecting it from the house floor plan. I want to selectively record camera output and replay camera output. I also want to be able to block access to one or more cameras with a specific password. I also want the option of seeing small windows that show views from all cameras and then be able to pick the one I want enlarged.

Jamie: Those are called thumbnail views.

Meredith: Okay, then I want thumbnail views of all the cameras. I also want the interface for the surveillance function to have the same look and feel as all other *SafeHome* interfaces. I want it to be intuitive, meaning I don't want to have to read a manual to use it.

Facilitator: Good job. Now, let's go into this function in a bit more detail . . .

The *SafeHome* home surveillance function (subsystem) discussed in the sidebar identifies the following functions (an abbreviated list) that are performed by the **homeowner** actor:

- Select camera to view.
- Request thumbnails from all cameras.
- Display camera views in a PC window.
- Control pan and zoom for a specific camera.
- Selectively record camera output.
- Replay camera output.
- Access camera surveillance via the Internet.

As further conversations with the stakeholder (who plays the role of a homeowner) progress, the requirements gathering team develops use cases for each of the functions noted. In general, use cases are written first in an informal narrative fashion. If more formality is required, the same use case is rewritten using a structured format similar to the one proposed in Chapter 5 and reproduced later in this section as a sidebar.

To illustrate, consider the function *access camera surveillance via the Internet—display camera views* **(ACS-DCV).** The stakeholder who takes on the role of the **homeowner** actor might write the following narrative:

Use case: Access camera surveillance via the Internet—display camera views (ACS-DCV)

Actor: homeowner

If I'm at a remote location, I can use any PC with appropriate browser software to log on to the *SafeHome Products* website. I enter my user ID and two levels of passwords and once I'm validated, I have access to all functionality for my installed *SafeHome* system. To access a specific camera view, I select "surveillance" from the major function buttons displayed. I then select "pick a camera" and the floor plan of the house is displayed. I then select the camera that I'm interested in. Alternatively, I can look at thumbnail snapshots from all cameras simultaneously by selecting "all cameras" as my viewing choice. Once I choose a camera, I select "view" and a one-frame-per-second view appears in a viewing window that is identified by the camera ID. If I want to switch cameras, I select "pick a camera" and the original viewing window disappears and the floor plan of the house is displayed again. I then select the camera that I'm interested in. A new viewing window appears.

A variation of a narrative use case presents the interaction as an ordered sequence of user actions. Each action is represented as a declarative sentence. Revisiting the **ACS-DCV** function, you would write:

Use case: Access camera surveillance via the Internet—display camera views (ACS-DCV)

Actor: homeowner

1. The homeowner logs onto the *SafeHome Products* website.

2. The homeowner enters his or her user ID.

3. The homeowner enters two passwords (each at least eight characters in length).

4. The system displays all major function buttons.

5. The homeowner selects the "surveillance" from the major function buttons.

6. The homeowner selects "pick a camera."

7. The system displays the floor plan of the house.

8. The homeowner selects a camera icon from the floor plan.

9. The homeowner selects the "view" button.

10. The system displays a viewing window that is identified by the camera ID.

11. The system displays video output within the viewing window at one frame per second.

It is important to note that this sequential presentation does not consider any alternative interactions (the narrative is more free-flowing and did represent a few alternatives). Use cases of this type are sometimes referred to as *primary scenarios* [Sch98a].

uote:

"Use cases can be used in many [software] processes. Our favorite is a process that is iterative and risk driven."

Geri Schneider and Jason Winters

6.2.2   Refining a Preliminary Use Case

A description of alternative interactions is essential for a complete understanding of the function that is being described by a use case. Therefore, each step in the primary scenario is evaluated by asking the following questions [Sch98a]:

- *Can the actor take some other action at this point?*
- *Is it possible that the actor will encounter some error condition at this point?* If so, what might it be?
- *Is it possible that the actor will encounter some other behavior at this point (e.g., behavior that is invoked by some event outside the actor's control)?* If so, what might it be?

Answers to these questions result in the creation of a set of *secondary scenarios* that are part of the original use case but represent alternative behavior. For example, consider steps 6 and 7 in the primary scenario presented earlier:

6. The homeowner selects "pick a camera."
7. The system displays the floor plan of the house.

Can the actor take some other action at this point? The answer is "yes." Referring to the free-flowing narrative, the actor may choose to view thumbnail snapshots of all cameras simultaneously. Hence, one secondary scenario might be "View thumbnail snapshots for all cameras."

Is it possible that the actor will encounter some error condition at this point? Any number of error conditions can occur as a computer-based system operates. In this context, we consider only error conditions that are likely as a direct result of the action described in step 6 or step 7. Again the answer to the question is "yes." A floor plan with camera icons may have never been configured. Hence, selecting "pick a camera" results in an error condition: "No floor plan configured for this house."[9] 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 at any time for virtually all interactions, it will not become part of the **ACS-DCV** use case. Rather, a separate use case—**Alarm condition encountered**—would be developed and referenced from other use cases as required.

---

9   In this case, another actor, the **system administrator,** would have to configure the floor plan, install and initialize (e.g., assign an equipment ID) all cameras, and test each camera to be certain that it is accessible via the system and through the floor plan.

Each of the situations described in the preceding paragraphs is characterized as a use-case exception. An *exception* describes a situation (either a failure condition or an alternative chosen by the actor) that causes the system to exhibit somewhat different behavior.

Cockburn [Coc01b] recommends using a "brainstorming" session to derive a reasonably complete set of exceptions for each use case. In addition to the three generic questions suggested earlier in this section, the following issues should also be explored:

- *Are there cases in which some "validation function" occurs during this use case?* This implies that validation function is invoked and a potential error condition might occur.

- *Are there cases in which a supporting function (or actor) will fail to respond appropriately?* For example, a user action awaits a response but the function that is to respond times out.

- *Can poor system performance result in unexpected or improper user actions?* For example, a Web-based interface responds too slowly, resulting in a user making multiple selects on a processing button. These selects queue inappropriately and ultimately generate an error condition.

The list of extensions developed as a consequence of asking and answering these questions should be "rationalized" [Co01b] using the following criteria: an exception should be noted within the use case if the software can detect the condition described and then handle the condition once it has been detected. In some cases, an exception will precipitate the development of another use case (to handle the condition noted).

6.2.3 Writing a Formal Use Case

The informal use cases presented in Section 6.2.1 are sometimes sufficient for requirements modeling. However, when a use case involves a critical activity or describes a complex set of steps with a significant number of exceptions, a more formal approach may be desirable.

The **ACS-DCV** use case shown in the sidebar follows a typical outline for formal use cases. The *goal in context* identifies the overall scope of the use case. The *precondition* describes what is known to be true before the use case is initiated. The *trigger* identifies the event or condition that "gets the use case started" [Coc01b]. The *scenario* lists the specific actions that are required by the actor and the appropriate system responses. *Exceptions* identify the situations uncovered as the preliminary use case is refined (Section 6.2.2). Additional headings may or may not be included and are reasonably self-explanatory.

SafeHome

Use Case Template for Surveillance

Use case: Access camera surveillance via the Internet—display camera views (ACS-DCV)

Iteration: 2, last modification: January 14 by V. Raman.

Primary actor: Homeowner.

Goal in context: To view output of camera placed throughout the house from any remote location via the Internet.

Preconditions: System must be fully configured; appropriate user ID and passwords must be obtained.

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

Scenario:

1. The homeowner logs onto the *SafeHome Products* website.
2. The homeowner enters his or her user ID.
3. The homeowner enters two passwords (each at least eight characters in length).
4. The system displays all major function buttons.
5. The homeowner selects the "surveillance" from the major function buttons.
6. The homeowner selects "pick a camera."
7. The system displays the floor plan of the house.
8. The homeowner selects a camera icon from the floor plan.
9. The homeowner selects the "view" button.
10. The system displays a viewing window that is identified by the camera ID.
11. The system displays video output within the viewing window at one frame per second.

Exceptions:

1. ID or passwords are incorrect or not recognized—see use case **Validate ID and passwords.**
2. Surveillance function not configured for this system—system displays appropriate error message; see use case **Configure surveillance function.**
3. Homeowner selects "View thumbnail snapshots for all camera"—see use case **View thumbnail snapshots for all cameras.**
4. A floor plan is not available or has not been configured—display appropriate error message and see use case **Configure floor plan.**
5. An alarm condition is encountered—see use case **Alarm condition encountered.**

Priority: Moderate priority, to be implemented after basic functions.

When available: Third increment.

Frequency of use: Moderate frequency.

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

Secondary actors: System administrator, cameras.

Channels to secondary actors:

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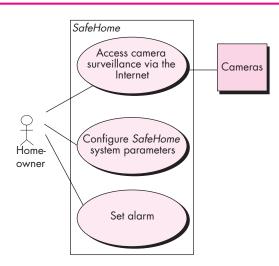
Open issues:

1. What mechanisms protect unauthorized use of this capability by employees of *SafeHome Products*?
2. Is security sufficient? Hacking into this feature would represent a major invasion of privacy.
3. Will system response via the Internet be acceptable given the bandwidth required for camera views?
4. Will we develop a capability to provide video at a higher frames-per-second rate when high-bandwidth connections are available?

In many cases, there is no need to create a graphical representation of a usage scenario. However, diagrammatic representation can facilitate understanding, particularly when the scenario is complex. As we noted earlier in this book, UML does provide use-case diagramming capability. Figure 6.4 depicts a preliminary use-case diagram for the *SafeHome* product. Each use case is represented by an oval. Only the **ACS-DCV** use case has been discussed in this section.

Every modeling notation has limitations, and the use case is no exception. Like any other form of written description, a use case is only as good as its author(s). If the description is unclear, the use case can be misleading or ambiguous. A use case focuses on functional and behavioral requirements and is generally inappropriate for nonfunctional requirements. For situations in which the requirements model must have significant detail and precision (e.g., safety critical systems), a use case may not be sufficient.

However, scenario-based modeling is appropriate for a significant majority of all situations that you will encounter as a software engineer. If developed properly, the use case can provide substantial benefit as a modeling tool.

6.3   UML MODELS THAT SUPPLEMENT THE USE CASE

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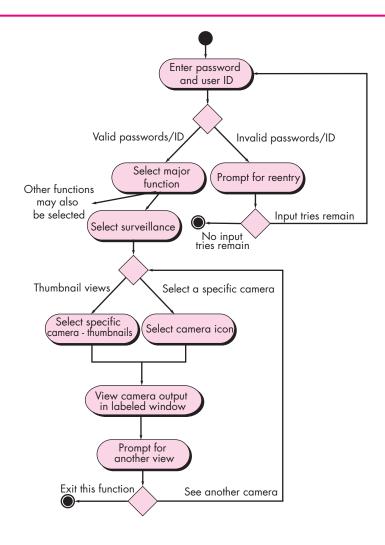
6.3.1   Developing an Activity Diagram

KEY POINT

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

The UML activity diagram supplements the use case by providing a graphical representation of the flow of interaction within a specific scenario. Similar to the flowchart, an activity diagram uses rounded rectangles to imply a specific system function, arrows to represent flow through the system, decision diamonds to depict a branching decision (each arrow emanating from the diamond is labeled), and solid horizontal lines to indicate that parallel activities are occurring. An activity diagram for the **ACS-DCV** use case is shown in Figure 6.5. It should be noted that the activity diagram adds additional detail not directly mentioned (but implied) by the use case.

For example, a user may only attempt to enter **userID** and **password** a limited num-
ber of times. This is represented by a decision diamond below "Prompt for reentry."
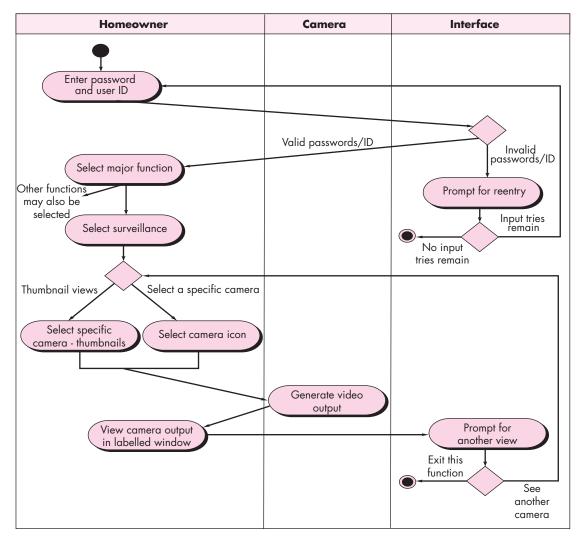
6.3.2   Swimlane Diagrams

The UML *swimlane diagram* is a useful variation of the activity diagram and allows
you to represent the flow of activities described by the use case and at the same time
indicate which actor (if there are multiple actors involved in a specific use case) or
analysis class (discussed later in this chapter) has responsibility for the action de-
scribed by an activity rectangle. Responsibilities are represented as parallel seg-
ments that divide the diagram vertically, like the lanes in a swimming pool.

Three analysis classes—**Homeowner, Camera,** and **Interface**—have direct or
indirect responsibilities in the context of the activity diagram represented in Figure 6.5.

FIGURE 6.6   Swimlane diagram for Access camera surveillance via the Internet—display camera views function

Referring to Figure 6.6, 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 reentry" and "prompt for another view." These prompts and the decisions associated with them fall within the **Interface** swimlane. However, arrows lead from that swimlane back to the **Homeowner** swimlane, where homeowner actions occur.

Use cases, along with the activity and swimlane diagrams, are procedurally oriented. They represent the manner in which various actors invoke specific functions

(or other procedural steps) to meet the requirements of the system. But a procedural view of requirements represents only a single dimension of a system. In Section 6.4, I examine the information space and how data requirements can be represented.

6.4 DATA MODELING CONCEPTS

If software requirements include the need to create, extend, or interface with a database or if complex data structures must be constructed and manipulated, the software team may choose to create a *data model* as part of overall requirements modeling. A software engineer or analyst defines all data objects that are processed within the system, the relationships between the data objects, and other information that is pertinent to the relationships. The *entity-relationship diagram* (ERD) addresses these issues and represents all data objects that are entered, stored, transformed, and produced within an application.

6.4.1 Data Objects

? How does a data object manifest itself within the context of an application?

A *data object* is a representation of composite information that must be understood by software. By *composite information,* I mean something that has a number of different properties or attributes. Therefore, width (a single value) would not be a valid data object, but **dimensions** (incorporating height, width, and depth) could be defined as an object.

A data object can be an external entity (e.g., anything that produces or consumes information), a thing (e.g., a report or a display), an occurrence (e.g., a telephone call) or event (e.g., an alarm), a role (e.g., salesperson), an organizational unit (e.g., accounting department), a place (e.g., a warehouse), or a structure (e.g., a file). For example, a **person** or a **car** can be viewed as a data object in the sense that either can be defined in terms of a set of attributes. The description of the data object incorporates the data object and all of its attributes.

KEY POINT

A data object is a representation of any composite information that is processed by software.

A data object encapsulates data only—there is no reference within a data object to operations that act on the data.[10] Therefore, the data object can be represented as a table as shown in Figure 6.7. The headings in the table reflect attributes of the object. In this case, a car is defined in terms of **make, model, ID number, body type, color,** and **owner**. The body of the table represents specific instances of the data object. For example, a Chevy Corvette is an instance of the data object **car.**

6.4.2 Data Attributes

KEY POINT

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

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

---

10 This distinction separates the data object from the class or object defined as part of the object-oriented approach (Appendix 2).

Tabular
representation
of data objects

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.

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**.

INFO

Data Objects and Object-Oriented 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[11] 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).

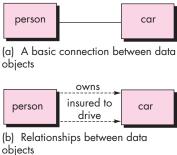An object-oriented class encapsulates data attributes but also incorporates the operations (methods) that manipulate the data implied by those attributes. In addition, the definition of classes implies a comprehensive infrastructure that is part of the object-oriented software engineering approach. Classes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

6.4.3 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

---

11 Readers who are unfamiliar with object-oriented concepts and terminology should refer to the brief tutorial presented in Appendix 2.

FIGURE 6.8

Relationships
between data
objects

(a) A basic connection between data
objects

(b) Relationships between data
objects

KEY POINT

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

illustrated in Figure 6.8a. A connection is established between **person** and **car** because the two objects are related. But what are the relationships? To determine the answer, you should understand the role of people (owners, in this case) and cars within the context of the software to be built. You can establish 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.

The relationships *owns* and *insured to drive* define the relevant connections between **person** and **car.** Figure 6.8b illustrates these object-relationship pairs graphically. The arrows noted in Figure 6.8b provide important information about the direction-ality of the relationship and often reduce ambiguity or misinterpretations.

INFO

Entity-Relationship Diagrams

The object-relationship pair is the cornerstone of the data model. These pairs can be represented graphically using the entity-relationship diagram (ERD).[12] The ERD was originally proposed by Peter Chen [Che77] for the design of relational database systems and has been extended by others. A set of primary components is identified for the ERD: data objects, attributes, relationships, and various type indicators. The primary purpose of the ERD is to represent data objects and their relationships.

Rudimentary ERD notation has already been introduced. Data objects are represented by a labeled rectangle. Relationships are indicated with a labeled line connecting objects. In some variations of the ERD, the connecting line contains a diamond that is labeled with the relationship. Connections between data objects and relationships are established using a variety of special symbols that indicate cardinality and modality.[13] If you desire further information about data modeling and the entity-relationship diagram, see [Hob06] or [Sim05].

---

12 Although the ERD is still used in some database design applications, UML notation (Appendix 1) can now be used for data design.
13 The *cardinality* of an object-relationship pair specifies "the number of occurrences of one [object] that can be related to the number of occurrences of another [object]" {Til93]. The *modality* of a re-lationship is 0 if there is no explicit need for the relationship to occur or the relationship is optional. The modality is 1 if an occurrence of the relationship is mandatory.

Data Modeling

Objective: Data modeling tools provide a software engineer with the ability to represent data objects, their characteristics, and their relationships. Used primarily for large database applications and other information systems projects, data modeling tools provide an automated means for creating comprehensive entity-relation diagrams, data object dictionaries, and related models.

Mechanics: Tools in this category enable the user to describe data objects and their relationships. In some cases, the tools use ERD notation. In others, the tools model relations using some other mechanism. Tools in this category are often used as part of database design and enable the creation of a database model by generating a database schema for common database management systems (DBMS).

Representative Tools:[14]

AllFusion ERWin, developed by Computer Associates (**www3.ca.com**), assists in the design of data objects, proper structure, and key elements for databases.

ER/Studio, developed by Embarcadero Software (**www.embarcadero.com**), supports entity-relationship modeling.

Oracle Designer, developed by Oracle Systems (**www.oracle.com**), "models business processes, data entities and relationships [that] are transformed into designs from which complete applications and databases are generated."

Visible Analyst, developed by Visible Systems (**www.visible.com**), supports a variety of analysis modeling functions including data modeling.

6.5 CLASS-BASED MODELING

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 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.

6.5.1 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, 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 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.

> **Quote:**
>
> "The really hard problem is discovering what are the right objects [classes] in the first place."
>
> **Carl Argila**

---

14 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.

But what should we look for once all of the nouns have been isolated? *Analysis classes* manifest themselves in one of the following ways:

- *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.[15] 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 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: "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, consider a grammatical parse (nouns are underlined, verbs italicized) for a processing narrative[16] for the *SafeHome* security function.

---

15 Another important categorization, defining entity, boundary, and controller classes, is discussed in Section 6.5.4.

16 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).

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 . . .

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 I call each entry in the list a potential object. You 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:

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.

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

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.

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).

6.5.2   Specifying Attributes

Attributes describe a class that has been selected for inclusion in the requirements 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?"

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 6.9).

Sensors are part of the overall *SafeHome* system, and yet they are not listed as data items or as attributes in Figure 6.9. **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.

6.5.3   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

FIGURE 6.9

Class diagram
for the system
class

System
systemID verificationPhoneNumber systemStatus delayTime telephoneNumber masterPassword temporaryPassword numberTries
program( ) display( ) reset( ) query( ) arm( ) disarm( )

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 6.5.5). 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, I explore this matter in a bit more detail.

SafeHome

Class Models

The scene: Ed's cubicle, as requirements 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 6.10.)

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 6.5.5.]

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.
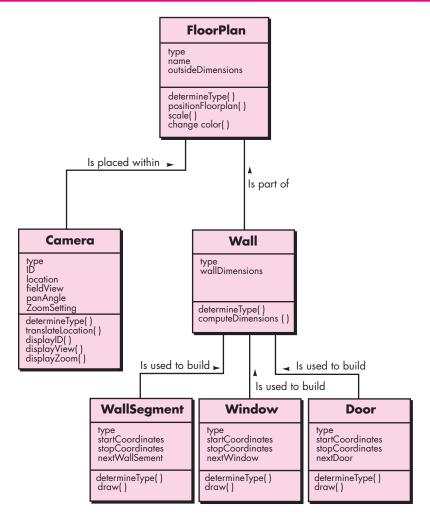

6.5.4 Class-Responsibility-Collaborator (CRC) Modeling

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

FIGURE 6.10

Class diagram for FloorPlan (see sidebar discussion)

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.

A simple CRC index card for the **FloorPlan** class is illustrated in Figure 6.11. 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 6.5.1 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**). These

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**

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

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 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" [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 infor-
mation 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 6.5.2 and 6.5.3. 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.[17] 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.[18] 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 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.

---

17 Cohesiveness is a design concept that is discussed in Chapter 8.
18 In such cases, it may be necessary to spit the class into multiple classes or complete subsystems in order to distribute intelligence more effectively.

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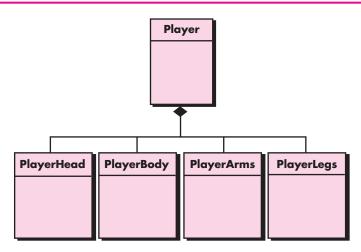
Collaborations are identified by determining whether a class can fulfill each responsibility itself. If it cannot, then it needs to interact with another class. Hence, a collaboration.

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 6.12.

FIGURE 6.12

A composite
aggregate
class

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.**

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 6.11).

When a complete CRC model has been developed, 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 **Sensor** 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, 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 6.10.)

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.

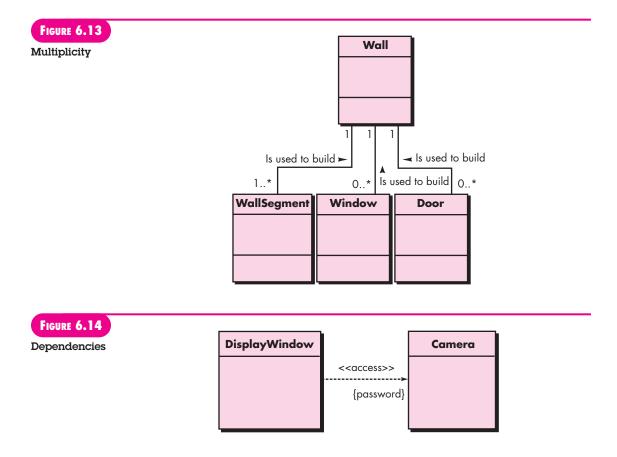**Ed:** Good way to find omissions or errors.

Vinod: Yep.

6.5.5   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 6.4.3). In UML these relationships are called *associations.* Referring back to Figure 6.10, 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*. Referring to Figure 6.10, 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 6.13, 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.[19]

---

19 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.

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 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 6.14 where <<access>> implies that the use of the camera output is controlled by a special password.
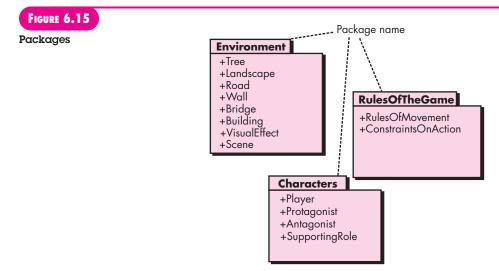
6.5.6 Analysis Packages

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 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 I 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,** 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 grouped in analysis packages as shown in Figure 6.15.

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.

FIGURE 6.15

Packages

6.6   SUMMARY

The objective of requirements modeling is to create a variety of representations that describe what the customer requires, establish a basis for the creation of a software design, and define a set of requirements that can be validated once the software is built. The requirements model bridges the gap between a system-level representation that describes overall system and business functionality and a software design that describes the software's application architecture, user interface, and component-level structure.

Scenario-based models depict software requirements from the user's point of view. The use case—a narrative or template-driven description of an interaction between an actor and the software—is the primary modeling element. Derived during requirements elicitation, the use case defines the keys steps for a specific function or interaction. The degree of use-case formality and detail varies, but the end result provides necessary input to all other analysis modeling activities. Scenarios can also be described using an activity diagram—a flowchart-like graphical representation that depicts the processing flow within a specific scenario. Swim-lane diagrams illustrate how the processing flow is allocated to various actors or classes.

Data modeling is used to describe the information space that will be constructed or manipulated by the software. Data modeling begins by representing data objects—composite information that must be understood by the software. The attributes of each data object are identified and relationships between data objects are described.

Class-based modeling uses information derived from scenario-based and data modeling elements 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, and dependencies among classes. Analysis packages are used to categorize and group classes in a manner that makes them more manageable for large systems.

PROBLEMS AND POINTS TO PONDER

6.1. Is it possible to begin coding immediately after an analysis model has been created? Explain your answer and then argue the counterpoint.

6.2. An analysis rule of thumb is that the model "should focus on requirements that are visible within the problem or business domain." What types of requirements are *not* visible in these domains? Provide a few examples.

6.3. What is the purpose of domain analysis? How is it related to the concept of requirements patterns?

6.4. Is it possible to develop an effective analysis model without developing all four elements shown in Figure 6.3? Explain.

6.5. You have been asked to build one of the following systems:

   a. a network-based course registration system for your university.
   b. a Web-based order-processing system for a computer store.
   c. a simple invoicing system for a small business.
   d. an Internet-based cookbook that is built into an electric range or microwave.

Select the system that is of interest to you and develop an entity-relationship diagram that describes data objects, relationships, and attributes.

6.6. The department of public works for a large city has decided to develop a Web-based pothole tracking and repair system (PHTRS). A description follows:

> Citizens can log onto a website and report the location and severity of potholes. As potholes are reported they are logged within a "public works department repair system" and are assigned an identifying number, stored by street address, size (on a scale of 1 to 10), location (middle, curb, etc.), district (determined from street address), and repair priority (determined from the size of the pothole). Work order data are associated with each pothole and include pothole location and size, repair crew identifying number, number of people on crew, equipment assigned, hours applied to repair, hole status (work in progress, repaired, temporary repair, not repaired), amount of filler material used, and cost of repair (computed from hours applied, number of people, material and equipment used). Finally, a damage file is created to hold information about reported damage due to the pothole and includes citizen's name, address, phone number, type of damage, and dollar amount of damage. PHTRS is an online system; all queries are to be made interactively.

   a. Draw a UML use case diagram for the PHTRS system. You'll have to make a number of assumptions about the manner in which a user interacts with this system.
   b. Develop a class model for the PHTRS system.

6.7. Write a template-based use case for the *SafeHome* home management system described informally in the sidebar following Section 6.5.4.

6.8. Develop a complete set of CRC model index cards on the product or system you chose as part of Problem 6.5.

6.9. Conduct a review of the CRC index cards with your colleagues. How many additional classes, responsibilities, and collaborators were added as a consequence of the review?

6.10. What is an analysis package and how might it be used?

Further Readings and Information Sources

Use cases can serve as the foundation for all requirements modeling approaches. The subject is discussed at length by Rosenberg and Stephens (*Use Case Driven Object Modeling with UML: Theory and Practice,* Apress, 2007), Denny (*Succeeding with Use Cases: Working Smart to Deliver Quality,* Addison-Wesley, 2005), Alexander and Maiden (eds.) (*Scenarios, Stories, Use Cases: Through the Systems Development Life-Cycle,* Wiley, 2004), Bittner and Spence (*Use Case Modeling,* Addison-Wesley, 2002), Cockburn [Coc01b], and other references noted in both Chapters 5 and 6.

Data modeling presents a useful method for examining the information space. Books by Hoberman [Hob06] and Simsion and Witt [Sim05] provide reasonably comprehensive treatments. In addition, Allen and Terry (*Beginning Relational Data Modeling,* 2d ed., Apress, 2005), Allen (*Data Modeling for Everyone,* Wrox Press, 2002), Teorey and his colleagues (*Database Modeling and Design: Logical Design,* 4th ed., Morgan Kaufmann, 2005), and Carlis and Maguire (*Mastering Data Modeling,* Addison-Wesley, 2000) present detailed tutorials for creating

industry-quality data models. An interesting book by Hay (*Data Modeling Patterns,* Dorset House, 1995) presents typical data model patterns that are encountered in many different businesses.

UML modeling techniques that can be applied for both analysis and design are discussed by O'Docherty (*Object-Oriented Analysis and Design: Understanding System Development with UML 2.0,* Wiley, 2005), Arlow and Neustadt (*UML 2 and the Unified Process,* 2d ed., Addison-Wesley, 2005), Roques (*UML in Practice,* Wiley, 2004), Dennis and his colleagues (*Systems Analysis and Design with UML Version 2.0,* Wiley, 2004), Larman (*Applying UML and Patterns,* 2d ed., Prentice-Hall, 2001), and Rosenberg and Scott (*Use Case Driven Object Modeling with UML,* Addison-Wesley, 1999).

A wide variety of information sources on requirements modeling are available on the Internet. An up-to-date list of World Wide Web references that are relevant to analysis modeling can be found at the SEPA website: **www.mhhe.com/engcs/compsci/pressman/ professional/olc/ser.htm**.