CHAPTER

7 UNDERSTANDING REQUIREMENTS

Understanding problem requirements is among the most difficult tasks facing a software engineer. When you first think about it, developing a clear understanding of the 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 they need? Surprisingly, in many instances, the answer to these questions is no. And even if customers and end users can state their needs explicitly, those needs will change throughout the project.

KEY CONCEPTS

analysis model 118
analysis patterns 122
collaboration 108
elaboration 104
elicitation 104
inception 104
negotiation 105
requirements engineering 103
requirements gathering 110
requirements management 106
requirements monitoring 123
specification 105
stakeholders 107
use cases 113
validating requirements 123
validation 105
viewpoints 107
work products 114

QUICK LOOK

What is it? Before you begin any technical work, it's a good idea to create a set of requirements for the engineering tasks. By establishing a set of requirements, you'll gain 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 and other project stakeholders (managers, customers, and end users) all participate in requirements engineering.

Why is it important? To understand what the customer wants before you begin to design and build a computer-based system. Building an elegant computer program that solves the wrong problem helps no one.

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 onward 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? Requirements engineering provides all parties with a written understanding of the problem. The work products may include: usage scenarios, function and feature lists, and requirements models.

How do I ensure that I've done it right? Requirements engineering work products are reviewed with stakeholders to ensure that everyone is on the same page. A word of warning: Even after all parties agree, things will change, and they will continue to change throughout the project.

In the forward to a book by Ralph Young [You01] on effective requirements practices, one of us [RSP] 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 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 we'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.

7.1 REQUIREMENTS ENGINEERING

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 that all else is secondary. What makes these arguments seductive is that they contain elements of truth. But each argument is flawed and can lead to a failed software project.

Requirements engineering is the term for the broad spectrum of tasks and techniques that lead to an understanding of requirements. 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. 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. It must be adapted to the needs of the process, the project, the product, and the people doing the work. It is important to realize that each of these tasks is done iteratively as the project team and the stakeholders continue to share information about their respective concerns.

Requirements engineering builds a bridge to design and construction. But where does the bridge originate? One could argue that it begins with the project stakeholders (e.g., managers, customers, and end users), where business needs are 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 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 resulting design. Requirements engineering encompasses seven tasks with sometimes muddy boundaries: *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. Expect to do a bit of design during requirements work and a bit of requirements work during design.

7.1.1 Inception

How does a software project get started? In general, most projects begin with an identified business need or when a potential new market or service is discovered. At project inception, you establish a basic understanding of the problem, the people who want a solution, and the nature of the solution that is desired. Communication between all stakeholders and the software team needs to be established during this task to begin an effective collaboration.

7.1.2 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.

An important part of elicitation is to understand the business goals [Cle10]. A *goal* is a long-term aim that a system or product must achieve. Goals may deal with either functional or nonfunctional (e.g., reliability, security, usability) concerns [Lam09].

Goals are often a good way to explain requirements to stakeholders and, once established, can be used to manage conflicts among stakeholders. Goals should be specified precisely and serve as the basis for requirements elaboration, verification and validation, conflict management, negotiation, explanation, and evolution.

Your job is to engage stakeholders and to encourage them to share their goals honestly. Once the goals are captured, you establish a prioritization mechanism and create a design rationale for a potential architecture (that meets stakeholder goals).

Agility is an important aspect of requirements engineering. The intent of elicitation is to transfer ideas from stakeholders to the software team smoothly and without delay. It is highly likely that new requirements will continue to emerge as iterative product development occurs.

7.1.3 Elaboration

The elaboration task focuses on developing a refined requirements model that identifies various aspects of software function, behavior, and information (Chapter 8). Elaboration is driven by the creation and refinement of user scenarios obtained during elicitation. These scenarios describe how the end users (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 that are required by each class are identified. The relationships and collaboration between classes are identified. Elaboration is a good thing, but you need to know when to stop. The key is to describe the problem in a way that establishes a firm base for design and then move on. Do not obsess over unnecessary details.

7.1.4 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."

These conflicts need to be reconciled through the process of negotiation. Customers, users, and other stakeholders are asked to rank requirements and then discuss conflicts in priority. 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. You should use an iterative approach that prioritizes requirements, assesses their cost and risk, and addresses internal conflicts. In this way, requirements are eliminated, combined, and/or modified so that each party achieves some measure of satisfaction.

7.1.5 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 consistent and therefore more understandable manner. However, it is sometimes necessary to remain flexible when a specification is to be developed. The formality and format of a specification varies with the size and the complexity of the software to be built. For large systems, a written document, combining natural language descriptions and graphical models, may be the best approach. A template for a formal software requirements specification document can be downloaded from: https://web.cs.dal.ca/~hawkey/3130/srs_template-ieee.doc. However, usage scenarios may be all that are required for smaller products or systems that reside within well-understood technical environments.

7.1.6 Validation

The work products produced during requirements engineering are assessed for quality during a validation step. A key concern during requirements validation is consistency. Use the analysis model to ensure that requirements have been consistently stated. Requirements validation examines the specification to ensure that all software requirements have been 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 16). 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.

To illustrate some of the problems that occur during requirements validation, consider two seemingly innocuous requirements:

- The software should be user friendly.
- The probability of a successful unauthorized database intrusion should be less than 0.0001.

The first requirement is too vague for developers to test or assess. What exactly does "user friendly" mean? To validate it, it must be quantified or qualified in some manner.

The second requirement has a quantitative element ("less than 0.0001"), but intrusion testing will be difficult and time consuming. Is this level of security even warranted for the application? Can other complementary requirements associated with security (e.g., password protection, specialized handshaking) replace the quantitative requirement noted?

7.1.7 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. Many of these activities are identical to the software configuration management (SCM) techniques discussed in Chapter 22.

INFO

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:

1. Are requirements stated clearly? Can they be misinterpreted?
2. 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?
3. Is the requirement bounded in quantitative terms?
4. What other requirements relate to this requirement? Are they clearly noted via a cross-reference matrix or other mechanism?
5. Does the requirement violate any system domain constraints?
6. Is the requirement testable? If so, can we specify tests (sometimes called validation criteria) to exercise the requirement?
7. Is the requirement traceable to any system model that has been created?
8. Is the requirement traceable to overall system and product objectives?
9. Is the specification structured in a way that leads to easy understanding, easy reference, and easy translation into more technical work products?
10. Has an index for the specification been created?
11. Have requirements associated with performance, behavior, and operational characteristics been clearly stated? What requirements appear to be implicit?

7.2 Establishing the Groundwork

In an ideal setting, stakeholders and software engineers work together on the same team. 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 reside in different cities or countries, 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 common, and you are often forced to work within the constraints imposed by this situation.

In the sections that follow, we 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.

7.2.1 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." We have already 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 7.3). The initial list will grow as stakeholders are contacted because every stakeholder will be asked: "Whom else do you think I should talk to?"

7.2.2 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 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.

Several things can make it hard to elicit requirements for software that satisfies its users: project goals are unclear, stakeholders' priorities differ, people have unspoken assumptions, stakeholders interpret meanings differently, and requirements are stated in a way that makes them difficult to verify [Ale11]. The goal of effective requirements engineering is to eliminate or at least reduce these problems.

7.2.3 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, we have noted that customers (and other stakeholders) should 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.

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.

INFO

Using "Planning Poker"

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 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.

7.2.4 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 and the overall project goals and benefits. For example, you might ask:

- 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 her perceptions about a solution:

- 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 (Q&A) 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 7.3.

7.2.5 Nonfunctional Requirements

A *nonfunctional requirement* (NFR) can be described as a quality attribute, a performance attribute, a security attribute, or a general constraint on a system. These are often not easy for stakeholders to articulate. Chung [Chu09] suggests that there is a lopsided emphasis on functionality of the software, yet the software may not be useful or usable without the necessary nonfunctional characteristics.

It is possible to define a two-phase approach [Hne11] that can assist a software team and other stakeholders in identifying nonfunctional requirements. During the first phase, a set of software engineering guidelines is established for the system to be built. These include guidelines for best practice, but also address architectural style (Chapter 10) and the use of design patterns (Chapter 14). A list of NFRs (e.g., requirements that address usability, testability, security, or maintainability) is then developed. A simple table lists NFRs as *column labels* and software engineering guidelines as *row labels*. A relationship matrix compares each guideline to all others, helping the team to assess whether each pair of guidelines is *complementary*, *overlapping*, *conflicting*, or *independent*.

In the second phase, the team prioritizes each nonfunctional requirement by creating a homogeneous set of nonfunctional requirements using a set of decision rules that establish which guidelines to implement and which to reject.

7.2.6 Traceability

Traceability is a software engineering term that refers to documented links between software engineering work products (e.g., requirements and test cases). A *traceability matrix* allows a requirements engineer to represent the relationship between requirements and other software engineering work products. Rows of the traceability

matrix are labeled using requirement names, and columns can be labeled with the name of a software engineering work product (e.g., a design element or a test case). A matrix cell is marked to indicate the presence of a link between the two.

The traceability matrices can support a variety of engineering development activities. They can provide continuity for developers as a project moves from one project phase to another, regardless of the process model being used. Traceability matrices often can be used to ensure the engineering work products have taken all requirements into account.

As the number of requirements and the number of work products grows, it becomes increasingly difficult to keep the traceability matrix up to date. Nonetheless, it is important to create some means for tracking the impact and evolution of the product requirements [Got11].

7.3 REQUIREMENTS GATHERING

Requirements gathering combines elements of problem solving, elaboration, negotiation, and specification. 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].

7.3.1 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 (either real or virtual) 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 worksheets, flip charts, or wall stickers or an electronic bulletin board, chat room, or virtual forum) is used.

The goal is to identify the problem, propose elements of the solution, negotiate different approaches, and specify a preliminary set of solution requirements.

A one- or two-page "product request" is generated during inception (Section 7.2). 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. If a system or product will serve many users, be absolutely certain that requirements are elicited from a representative cross section of users. If only one user defines all requirements, acceptance risk is high (meaning there may be several other stakeholders who will not accept the product). The product request is distributed to all attendees before the meeting date.

As an example, 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. We might also consider using voice control of the system using some technology like Alexa.
>
> 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, can be programmed by the homeowner, and will automatically contact a monitoring agency and the owner's cell phone 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 is present, omissions are likely to 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, security) 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.

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 tablet, 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 using a wireless router, *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 1 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 a group forum or at an internal website or posed in a social networking 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 deleted, and additions can be made. At this stage, critique and debate are strictly prohibited. Avoid the impulse to shoot down a customer's idea as "too costly" or "impractical." The idea here is to negotiate a list that is acceptable to all. To do this, you must keep an open mind.

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 or 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 or by creating a use case (Section 7.4) that involves the 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 230 × 130 mm in size. The control panel has wireless connectivity to sensors and a tablet. User interaction occurs through a keypad containing 12 keys. A 75 × 75 mm OLED 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.

SAFEHOME

Case Study Example
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.)

Many stakeholder concerns (e.g., accuracy, data accessibility, security) are the basis for nonfunctional system requirements (Section 7.2). As stakeholders enunciate these concerns, software engineers must consider them within the context of the system to be built. The questions that must be answered [Lag10] are:

- Can we build the system?
- Will this development process allow us to beat our competitors to market?
- Do adequate resources exist to build and maintain the proposed system?
- Will the system performance meet the needs of our customers?

7.3.2 Usage Scenarios

As requirements are gathered, an overall vision of system functions and features begin to materialize. However, it is difficult to move into more technical software engineering activities until you understand how the 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 7.4.

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.)

7.3.3 Elicitation Work Products

The work products produced during requirements elicitation will vary depending on the size of the system or product to be built. For large systems, the work products may include: (1) a statement of need and feasibility; (2) a bounded statement of scope for the system or product; (3) a list of customers, users, and other stakeholders who participated in requirements elicitation; (4) a description of the system's technical environment; (5) a list of requirements (preferably organized by function) and the domain constraints that apply to each; and (6) a set of usage scenarios that provide insight into the use of the system or product under different operating conditions. Each of these work products is reviewed by all people who have participated in requirements elicitation.

7.4 Developing Use Cases

A use case tells a stylized story about how an end user (playing one of several possible roles) interacts with the system under a specific set of circumstances. The story may be narrative text (a *user story*), 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 will 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 several 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 user who interacts with the program that allows experimenting with alarm sensor configuration

in a virtual building. After careful review of requirements, the software for the control computer requires four different modes (roles) for interaction: placement mode, testing mode, monitoring mode, and troubleshooting mode. Therefore, four actors can be defined: editor, tester, monitor, and troubleshooter. In some cases, the user can play all the 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 questions that should be answered by a use case:

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

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 different ways using either the alarm control panel, a tablet, or a cell phone.

The homeowner:

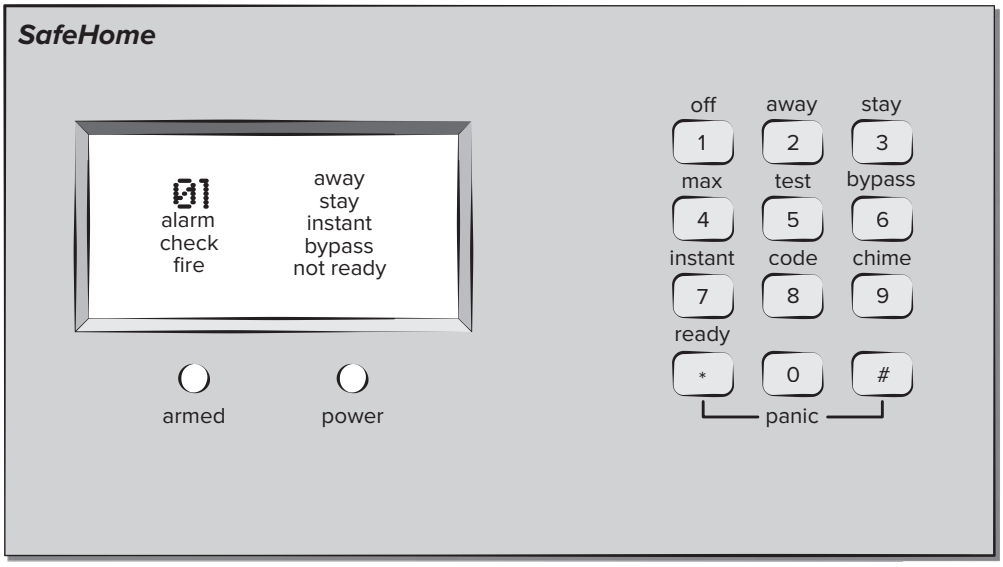
1. Enters a password to allow all other interactions
2. Inquires about the status of a security zone
3. Inquires about the status of a sensor
4. Presses the panic button in an emergency
5. Activates and deactivates the security system

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

1. The homeowner observes the *SafeHome* control panel (Figure 7.1) to determine if the system is ready for input. If the system is not ready, a *not ready*

FIGURE 7.1

SafeHome **control panel**



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.)

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 7.1) 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 user 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, that is, 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 red 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 and then 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. Use cases are often written informally as user stories. However, using the template shown here helps to ensure that you've addressed all key issues. This is very important for systems where user safety or security is a stakeholder concern.

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 7.2.)

Jamie: I'm just beginning to learn UML notation. 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.)

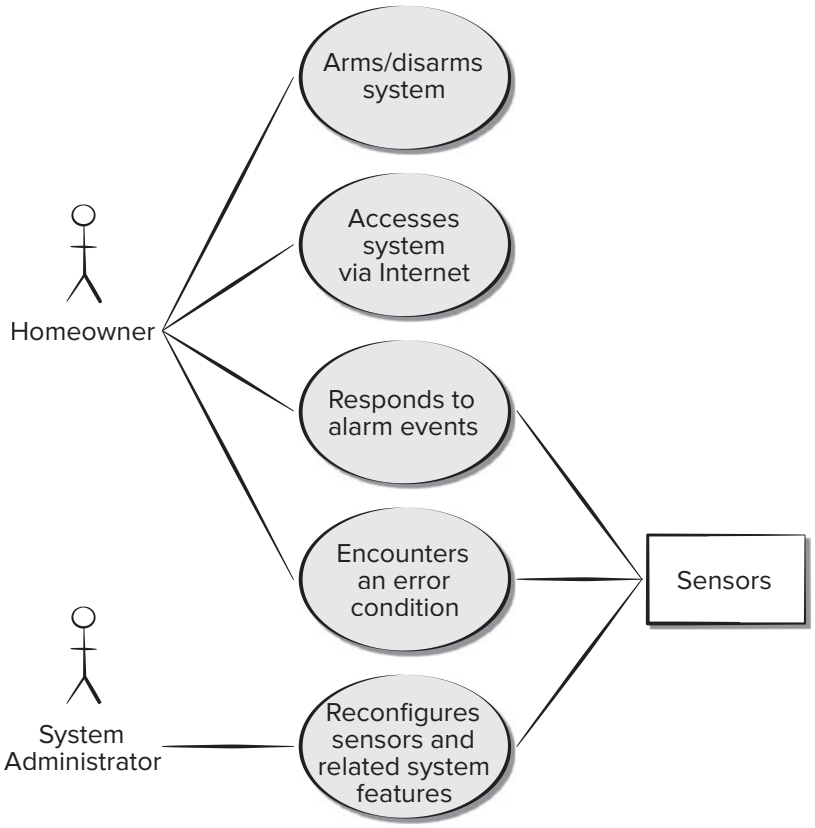
7.5 BUILDING THE ANALYSIS MODEL

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 as 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.

As the analysis 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. If your team finds that it does not use certain elements of the analysis model as the project moves to design and construction, those elements should not be created in the future and should not be maintained as the requirements change in the current project. The analysis model and the methods that are used to build it are presented in detail in Chapter 8. We present a brief overview in the sections that follow.

FIGURE 7.2

UML use case diagram for *SafeHome* home security function



7.5.1 Elements of the Analysis Model

There are many 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 several different modes of representation to depict the analysis model. Using different modes of representation forces you to consider requirements from different viewpoints—an approach that has a higher probability of uncovering omissions, inconsistencies, and ambiguity. It is always a good idea to get stakeholders involved. One of the best ways to do this is to have each stakeholder write use cases that describe how the software will be used. A set of generic elements common to most analysis models is introduced in this chapter.

Scenario-Based Elements. Scenario-based elements of the requirements model are often the first part of the model that is developed. They describe the system from the user's point of view. For example, basic user stories (Section 7.4) and their corresponding use case diagrams (Figure 7.2) may evolve into more elaborate template-based use cases (Section 7.4). As such, they serve as input for the creation of other modeling elements. It is always a good idea to get stakeholders involved. One of the best ways to do this is to have each stakeholder write use cases that describe how the software will be used.

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 7.3).

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. Other analysis modeling elements depict how classes collaborate with one another and the relationships and interactions among classes. One way to isolate classes is to look for descriptive nouns in a use case script. At least some of the nouns will be candidate classes. The verbs found in the use case script may be considered candidate methods for these classes. These and other techniques are discussed in more detail in Chapter 8.

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

The *state diagram* is one method for representing the behavior of a system by depicting its states and the events that cause the system to change state. A *state* is any externally observable mode of behavior. In addition, the state diagram indicates what actions (e.g., process activations) are taken when events occur. External stimuli (events) cause transitions between states.

FIGURE 7.3
Class diagram for sensor

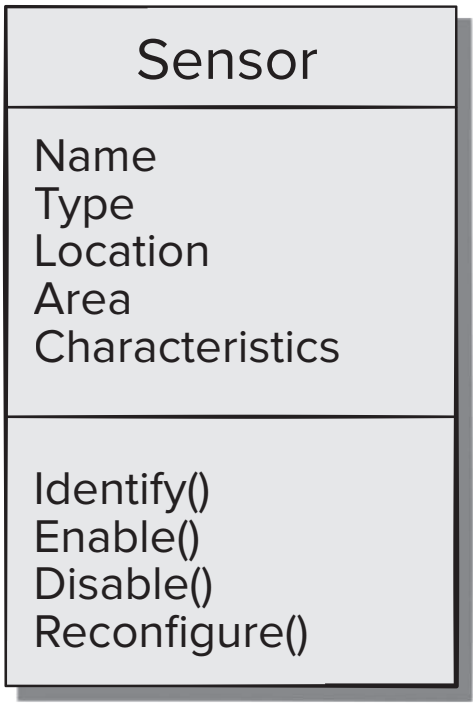
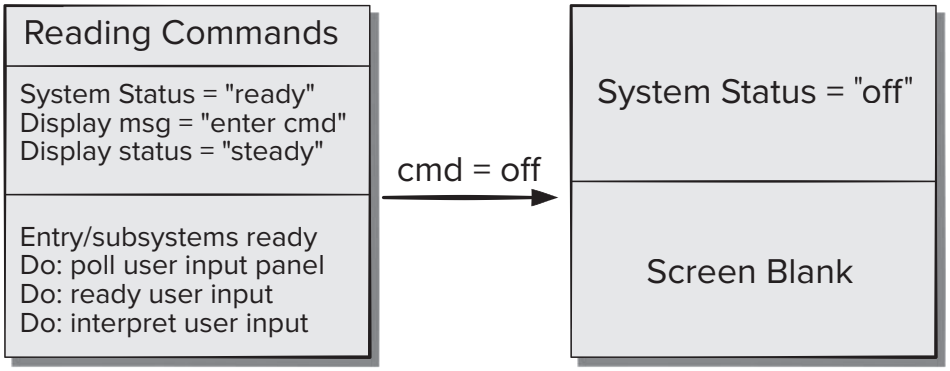


FIGURE 7.4

UML state diagram notation



To illustrate the use of a state diagram, consider software embedded within the *SafeHome* control panel that is responsible for reading user input. An example of UML state diagram notation is shown in Figure 7.4. Further discussion of behavioral modeling is presented in Chapter 8.

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.

7.5.2 Analysis Patterns

Anyone who has done requirements engineering on more than a few software projects notices that certain problems reoccur across all projects within a specific application domain. 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.

Analysis patterns are integrated into the analysis model by reference to the pattern name. They are also stored in a repository so that requirements engineers can use search facilities to find and reuse them. Information about an analysis pattern (and other types of patterns) is presented in a standard template [Gey01] that is discussed in more detail in Chapter 14. Examples of analysis patterns and further discussion of this topic are presented in Chapter 8.

7.6 NEGOTIATING REQUIREMENTS

In an ideal world, the requirements engineering tasks (inception, elicitation, and elaboration) determine customer requirements in sufficient detail to proceed to subsequent software engineering activities. Unfortunately, this rarely happens. You may have to enter into *negotiations* 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 these negotiations is to develop a project plan that meets stakeholder needs while at the 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. That is, stakeholders win by getting the system or product that satisfies most their needs, and you (as a member of the software team) win by working to realistic and achievable budgets and deadlines.

Fricker [Fri10] and his colleagues suggest replacing the traditional handoff of requirements specifications to software teams with a bidirectional communication process called handshaking. Handshaking might be one way to accomplish a win-win result. In *handshaking*, the software team proposes solutions to requirements, describes their impact, and communicates their intentions to customer representatives. The customer representatives review the proposed solutions, focusing on missing features and seeking clarification of novel requirements. Requirements are determined to be *good enough* if the customers accept the proposed solution. Handshaking tends to improve identification, analysis, and selection of variants and promotes win-win negotiation.

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.

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?

7.7 REQUIREMENTS MONITORING

Incremental development is commonplace. This means that use cases evolve, new test cases are developed for each new software increment, and continuous integration of source code occurs throughout a project. *Requirements monitoring* can be extremely useful when incremental development is used. It encompasses five tasks: (1) *distributed debugging* uncovers errors and determines their cause, (2) *run-time verification* determines whether software matches its specification, (3) *run-time validation* assesses whether the evolving software meets user goals, (4) *business activity monitoring* evaluates whether a system satisfies business goals, and (5) *evolution and codesign* provides information to stakeholders as the system evolves.

Incremental development implies the need for incremental validation. Requirements monitoring supports continuous validation by analyzing user goal models against the system in use. For example, a monitoring system might continuously assess user satisfaction and use feedback to guide incremental improvements [Rob10].

7.8 VALIDATING REQUIREMENTS

As each element of the requirements model is created, it is examined for inconsistency, omissions, and ambiguity. This is true even for agile process models where requirements tend to be written as user stories and/or test cases. The requirements represented

by the model are prioritized by stakeholders and grouped within requirements packages that will be implemented as software increments. A review of the requirements model addresses the following questions:

1. Is each requirement consistent with the overall objectives for the system or product?
2. 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?
3. Is the requirement really necessary or does it represent an add-on feature that may not be essential to the objective of the system?
4. Is each requirement bounded and unambiguous?
5. Does each requirement have attribution? That is, is a source (generally, a specific individual) noted for each requirement?
6. Do any requirements conflict with other requirements?
7. Is each requirement achievable in the technical environment that will house the system or product?
8. Is each requirement testable, once implemented?
9. Does the requirements model properly reflect the information, function, and behavior of the system to be built?
10. Has the requirements model been "partitioned" in a way that exposes progressively more detailed information about the system?
11. 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.

7.9 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 requirements engineering activities—inception, elicitation, elaboration, negotiation, specification, validation, and management—are conducted by members of the software team and product stakeholders.

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 and the development of usage scenarios (user stories).

Elaboration further expands requirements in a model—a collection of scenario-based, activity-based, class-based, and behavioral elements. The model may reference

analysis patterns, characteristics of the problem domain 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. Each requirement needs to be validated against customer needs to ensure that the right system is to be built.

PROBLEMS AND POINTS TO PONDER

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

7.2. 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?

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

7.4. Your instructor will divide the class into groups of four or 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.

7.5. 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. Searching for books (on a specific topic) using an online bookstore

7.6. Write a user story for one of the activities listed in Problem 7.5.

7.7. Consider the use case you created in Problem 7.5, and write a nonfunctional requirement for the application.

7.8. Using the template presented in Section 7.5.2, suggest one or more analysis patterns for the following application domains:

a. E-mail software.
b. Internet browsers.
c. Mobile app creation software.

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

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