CHAPTER

8

UNDERSTANDING REQUIREMENTS

KEY CONCEPTS

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, 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.

QUICK LOOK

What is it? Before you begin any technical work, it's a good idea to create a set of requirements for any 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, and 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 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? 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.

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.

8.1 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

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.[1] But each argument 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.

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.

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, and 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

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.

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 resultant design.

Over the past decade, there have been many technology changes that impact the requirements engineering process [Wev11]. Ubiquitous computing allows computer technology to be integrated into many everyday objects. When these objects are networked they can allow the creation of more complete user profiles, with the accompanying concerns for privacy and security.

Expect to do a bit of design during requirements work and a bit of requirements work during design.

Widespread availability of applications in the electronic marketplace will lead to more diverse stakeholder requirements. Stakeholders can customize a product to meet specific, targeted requirements that are applicable to only a small subset of all end users. As product development cycles shorten, there are pressures to streamline requirements engineering so that products come to market more quickly. But the fundamental problem remains the same, getting timely, accurate, and stable stakeholder input.

Requirements engineering 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.

"The seeds of major software disasters are usually sown in the first three months of commencing the software project."

Caper Jones

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,

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: www.mhhe.com/pressman

3 Recall that the Unified Process (Chapter 4) defines a more comprehensive "inception phase" that encompasses the inception, elicitation, and elaboration tasks discussed in this chapter.

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.

An important part of elicitation is to establish business goals [Cle10]. Your job is to engage stakeholders and to encourage them to share their goals honestly. Once the goals have been captured, a prioritization mechanism should be established, and a design rationale for a potential architecture (that meets stakeholder goals) can be created.

INFO

Goal-Oriented Requirements Engineering

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, etc.) concerns. Goals are often a good way to explain requirements to stakeholders and, once established, can be used to manage conflicts among stakeholders.

Object models (Chapters 10 and 11) and requirements can be derived systematically from goals. A goal graph showing links among goals can provide some degree of traceability (Section 8.2.6) between high-level strategic concerns to low-level technical details. Goals should be specified precisely and serve as the basis for requirements elaboration, verification/validation, conflict management, negotiation, explanation, and evolution.

Conflicts detected in requirements are often a result of conflicts present in the goals themselves. Conflict resolution is achieved by negotiating a set of mutually agreed-upon goals that are consistent with one another and with stakeholder desires. A more complete discussion on goals and requirements engineering can be found in a paper by Lamsweweerde [LaM01b].

? Why is it difficult to gain a clear understanding of what the customer wants?

Christel and Kang [Cri92] identify a number of problems that are encountered as elicitation occurs. *Problems of scope* occur when the boundary of the system is ill-defined or the customers and users specify unnecessary technical detail that may confuse, rather than clarify, overall system objectives. *Problems of understanding* are encountered when customers and 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, omit information that is believed to be "obvious," specify requirements that conflict with the needs of other customers and users, or specify requirements that are ambiguous or untestable. *Problems of volatility* occur when the requirements change over time. To help

overcome these problems, you must approach the requirements-gathering activity in an organized manner.

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.

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 9 through 11) 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.

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.

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.

KEY POINT

The formality and format of a specification varies with the size and the complexity of the software to be built.

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. 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.

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.

INFO

Software Requirements Specification Template

A *software requirements specification* (SRS) is a work product 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 in this sidebar.

A key concern during requirements validation is consistency. Use the analysis model to ensure that requirements have been consistently stated.

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 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 20). 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

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.

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?

Glinz [Gli09] writes that quality requirements need to be represented in a manner that delivers optimal value. This means assessing the risk (Chapter 35) of delivering a system that fails to meet the stakeholders' quality requirements and attempting to mitigate this risk at minimum cost. The more critical the quality requirement is, the greater the need to state it in quantifiable terms. Less-critical quality requirements can be stated in general terms. In some cases, a general quality requirement can be verified using a qualitative technique (e.g., user survey or check list). In other situations, quality requirements can be verified using a combination of qualitative and quantitative assessment.

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:

- 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 29.

SOFTWARE TOOLS

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 Volvere Requirements resources site at **www.volere.co.uk/tools.htm**. Requirements modeling tools are discussed in Chapters 9 and 10. Tools noted below focus on requirement management.

EasyRM, developed by Cybernetic Intelligence GmbH (**http://www.visuresolutions.com/visure-requirements-software**), Visure Requirements is a flexible and complete requirements engineering life-cycle solution, supporting requirements capture, analysis, specification, validation and verification, management, and reuse.

Rational RequisitePro, developed by Rational Software (**www-03.ibm.com/software/products/us/en/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 Volvere site noted earlier and at **www.jiludwig.com/Requirements_Management_Tools.html**.

8.2 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

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

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.

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, 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.

8.2.1 Identifying Stakeholders

KEY POINT

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

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

8.2.2 Recognizing Multiple Viewpoints

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

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.

There are several things that 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.

8.2.3 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.

INFO

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.

Quote:

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

James Thurber

8.2.4 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:

- 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?

Quote:

"He who asks a question is a fool for five minutes; he who does not ask a question is a fool forever."

Chinese proverb

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 8.3.

8.2.5 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 non-functional characteristics.

In Section 8.3.2, we discuss a technique called *quality function deployment* (QFD). Quality function deployment attempts to translate unspoken customer

needs or goals into system requirements. Nonfunctional requirements are often listed separately in a software requirements specification.

As an adjunct to QFD, 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 13) and the use of design patterns (Chapter 16). 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 [Hne11] that establish which guidelines to implement and which to reject.

8.2.6 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].

8.3 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]

8.3.1 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:

? What are the basic guidelines for conducting a collaborative requirements gathering meeting?

- 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 work sheets, flip charts, or wall stickers or an electronic bulletin board, chat room, or virtual forum) is used.

WebRef

Joint Application Development (JAD) is a popular technique for requirements gathering. A good description can be found at **www.carolla.com/wp-jad.htm.**

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 8.2). 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. 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, can be programmed by the homeowner, and will automatically telephone a monitoring agency when a situation is detected.

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

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.

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.

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) 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.

"Facts do not cease to exist because they are ignored."

Aldous Huxley

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).

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.

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, 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.

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 8.4) 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 x 130 mm in size. The control panel has wireless connectivity to sensors and a PC. User interaction occurs through a keypad containing 12 keys. A 75 x 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

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 8.2). As stakeholders enunciate these concerns, software engineers must consider them within the context

of the system to be built. Among the questions that must be answered [Lag10] are as follows:

- 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?

The answers to these and other questions will evolve over time.

8.3.2 Quality Function Deployment

KEY POINT

QFD defines requirements in a way that maximizes customer satisfaction.

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.

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!

Within the context of QFD, *normal requirements* identify 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. *Expected 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. *Exciting requirements* go beyond the customer's expectations and prove to be very satisfying when present.

WebRef

Useful information on QFD can be obtained at **www.qfdi.org.**

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].

8.3.3 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 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 8.4.

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.)

8.3.4 Elicitation Work Products

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

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: (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 applies to each, (6) a set of usage scenarios that provide insight into the use of the system or product under different operating conditions, and (7) any prototypes developed to better

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

8.3.5 Agile Requirements Elicitation

KEY POINT

User stories are the way to document requirements elicited from customers in agile process models.

Within the context of an agile process, requirements are elicited by asking all stakeholders to create *user stories*. Each user story describes a simple system requirement written from the user's perspective. User stories can be written on small note cards, making it easy for developers to select and manage a subset of requirements to implement for the next product increment. Proponents claim that using note cards written in the user's own language allows developers to shift their focus to communication with stakeholders on the selected requirements rather than their own agenda [Mai10a].

Although the agile approach to requirements elicitation is attractive for many software teams, critics argue that a consideration of overall business goals and nonfunctional requirements is often lacking. In some cases, rework is required to accommodate performance and security issues. In addition, user stories may not provide a sufficient basis for system evolution over time

8.3.6 Service-Oriented Methods

? What is a *service* in the context of service-oriented methods?

Service-oriented development views a system as an aggregation of services. A *service* can be "as simple as providing a single function, for example, a request/response-based mechanism that provides a series of random numbers, or can be an aggregation of complex elements, such as the Web service API" [Mic12].

Requirements elicitation in service-oriented development focuses on the definition of services to be rendered by an application. As a metaphor, consider the service provided when you visit a fine hotel. A doorperson greets guests. A valet parks their cars. The desk clerk checks the guests in. A bellhop manages the bags. The concierge assists guest with local arrangements. Each contact or *touchpoint* between a guest and a hotel employee is designed to enhance the hotel visit and represents a service offered.

KEY POINT

Requirements elicitation for service-oriented methods fines services render by an app. A touchpoint *represents an opportunity for the user to interact with the system to receive a desired service.*

Most service design methods emphasize understanding the customer, thinking creatively, and building solutions quickly [Mai10b]. To achieve these goals, requirements elicitation can include ethnographic studies,[11] innovation workshops, and early low-fidelity prototypes. Techniques for eliciting requirements must also acquire information about the brand and the stakeholders' perceptions of it. In addition to studying how the brand is used by customers, analysts need strategies to discover and document requirements about the desired qualities of new user experiences. User stories are helpful in this regard.

11 Studying user behavior in the environment where the proposed software product will be used.

The requirements for touchpoints should be characterized in a manner that indicates achievement of the overall service requirements. This suggests that each requirement should be traceable to a specific service.

8.4 Developing Use Cases

Use cases are defined from an actor's point of view. An actor is a role that people (users) or devices play as they interact with the software.

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.

WebRef

An excellent paper on use cases can be downloaded from **www.ibm.com/developerworks/webservices/library/co-design7.html.**

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.

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

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:

- 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?

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. 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, and (5) 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]

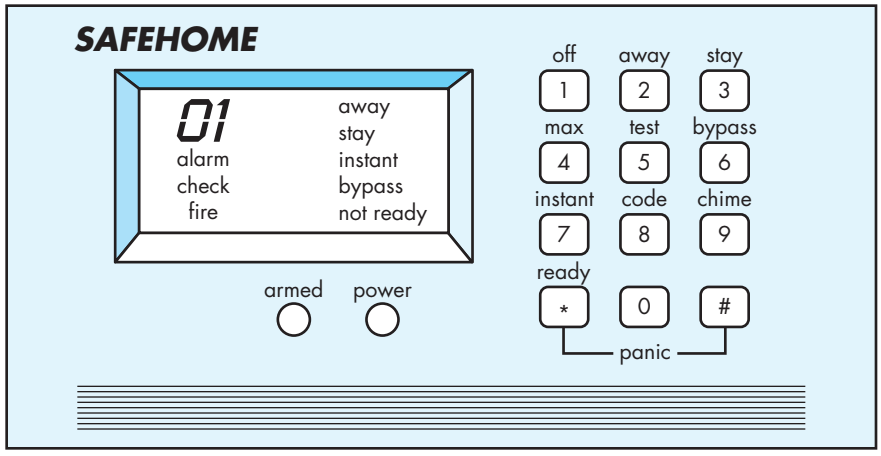
1. The homeowner observes the *SafeHome* control panel (Figure 8.1) to determine if the system is ready for input. If the system is not ready, a *not ready* message is displayed on the LCD display, and the homeowner must physically close windows or doors so that the *not ready* message disappears. [A *not ready* message implies that a sensor is open; i.e., that a door or window is open.]

12 Jacobson's questions have been extended to provide a more complete view of use case content.

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 GUI provided when a PC or mobile device is used.

FIGURE 8.1

SafeHome **control panel**



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 8.1) 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 cases are often written informally. However, use the template shown here to ensure that you've addressed all key issues.

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 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 8.2.)

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.

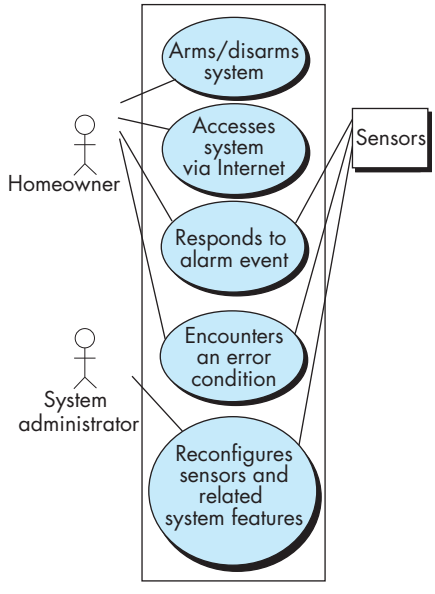
Marketing person: Wait, I've been looking at this diagram and all of a sudden I realize we missed something.

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

(The meeting continues.)

FIGURE 8.2

UML use case diagram for *SafeHome* home security function



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

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.

8.5 BUILDING THE ANALYSIS 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.

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. The analysis model and the methods that are used to build it are presented in detail in Chapters 9 to 11. We present a brief overview in the sections that follow.

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.

8.5.1 Elements of the Analysis Model

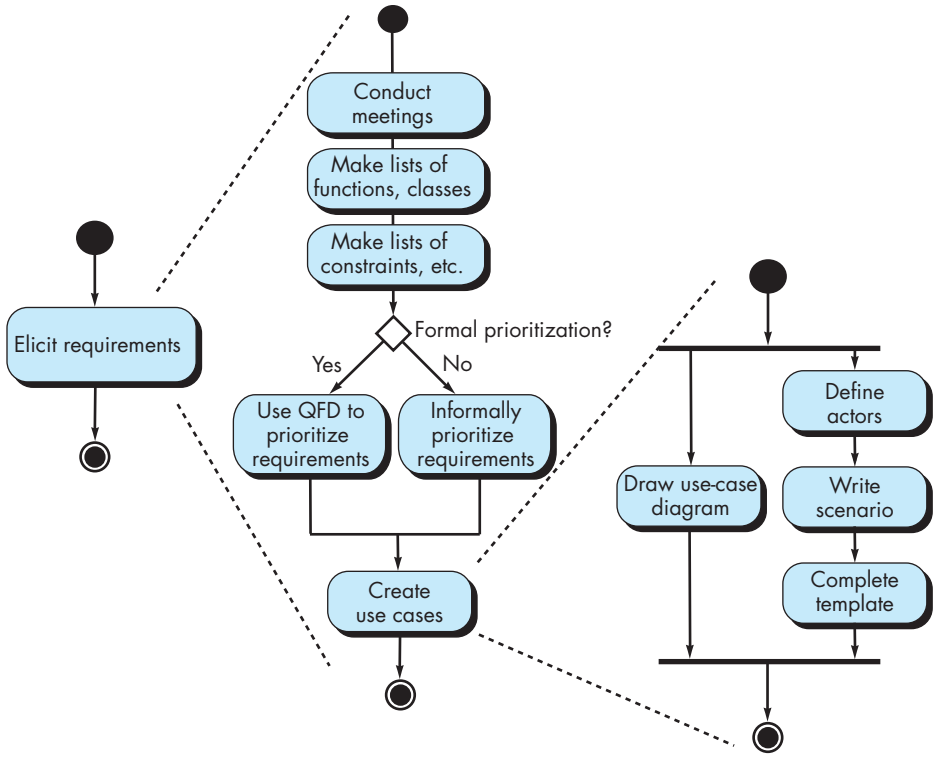
There are many different ways to look at the requirements for a computer-based system. Some software people argue that it's best to select one mode of representation (e.g., the use case) and apply it to the exclusion of all other modes. Other practitioners believe that it's worthwhile to use a number of different modes of representation to depict the analysis 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. A set of generic elements is common to most analysis models.

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, we use the terms *analysis model* and *requirements model* synonymously. Both refer to representations of the information, functional, and behavioral domains that describe problem requirements.

FIGURE 8.3

UML activity diagrams for eliciting requirements



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 8.4) and their corresponding use case diagrams (Figure 8.2) 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 8.3 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.

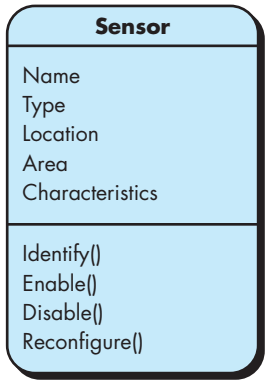
ADVICE

One way to isolate classes is to look for descriptive nouns in a use case script. At least some of the nouns will be candidate classes. More on this in Chapter 12.

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 8.4). 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

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

FIGURE 8.4
Class diagram for sensor



one another and the relationships and interactions between classes. These are discussed in more detail in Chapter 10.

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

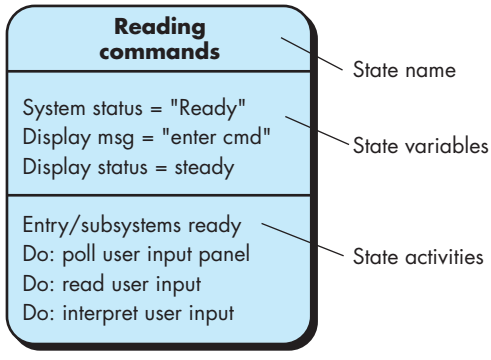
Behavioral elements. The behavior of a computer-based system can have a profound effect on the design that is chosen and the implementation approach that is applied. Therefore, the requirements model must provide modeling elements that depict behavior.

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 observable mode of behavior. In addition, the state diagram indicates what actions (e.g., process activation) are taken as a consequence of a particular event.

To illustrate 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 8.5.

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 11.

FIGURE 8.5
UML state diagram notation



SafeHome

Preliminary Behavioral Modeling

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

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

The conversation:

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

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

Ed (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.

8.5.2 Analysis Patterns

If you want to obtain solutions to customer requirements more rapidly and provide your team with proven approaches, use 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

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.

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][19] that is discussed in more detail in Chapter 16. Examples of analysis patterns and further discussion of this topic are presented in Chapter 11.

8.5.3 Agile Requirements Engineering

The intent of agile requirements engineering is to transfer ideas from stakeholders to the software team rather than create extensive analysis work products. In many situations, requirements are not predefined but emerge as each iteration of product development begins. As the agile team acquires a high-level understanding of a product's critical features use stories (Chapter 5) relevant to the next product increment are refined. The agile process encourages the early identification and implementation of the highest priority product features. This allows the early creation and testing of working prototypes.

Agile requirements engineering addresses important issues that are common in software projects: high requirements volatility, incomplete knowledge of development technology, and customers not able to articulate their visions until they see a working prototype. The agile process interleaves requirements engineering and design activities.

8.5.4 Requirements for Self-Adaptive Systems

Self-adaptive systems[20] can reconfigure themselves, augment their functionality, protect themselves, recover from failure, and accomplish all of this while hiding most of their internal complexity from their users [Qur09]. Adaptive requirements document the variability needed for self-adaptive systems. This means that a requirement must encompass the notion of variability or flexibility while at the same time specifying either a functional or quality aspect of the software product. Variability might include timing uncertainty, user profile differences (e.g., end users versus systems administrators), behavior changes based on problem domain (e.g., commercial or educational), or predefined behaviors exploiting system assets.

? What are the characteristics of a self-adaptive system?

Capturing adaptive requirements focuses on the same questions that are used for requirements engineering of more conventional systems. However, significant variability can be present when answering each of these questions. The more variable the answers, the more complex the resulting system will need to be to accommodate the requirements.

---

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.

20 An example of a self-adaptive system is a "location aware" app that adapts its behavior to the location of the mobile platform on which it resides.

8.6 Negotiating Requirements

Quote:

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

Ludwig Erhard

WebRef

A brief paper on negotiation for software requirements can be downloaded from **www.alexander-egyed.com/publications/Software_Requirements_Negotiation-Some_Lessons_Learned.html.**

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 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.[21] 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.

21 Dozens of books have been written on negotiating skills (e.g., [Fis11], [Lew09], [Rai06]). It is one of the more important skills that you can learn. Read one.

Fricker [Fri10] and his colleagues suggest replacing the traditional handoff of requirements specifications to software teams with a bidirectional communication process called *handshaking*. 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 allows detailed requirements to be delegated to software teams. The teams need to elicit requirements from customers (e.g., product users and domain experts), thereby improving product acceptance. 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?

8.7 REQUIREMENTS MONITORING

Today, 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].

8.8 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 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 or 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?

CHAPTER

9 Requirements Modeling: Scenario-Based Methods

Key Concepts

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

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 a scenario-based perspective and examine how UML can be used to supplement the scenarios. In Chapters 10 and 11, you'll learn about other "dimensions" of the requirements model. By examining a number of different dimensions, you'll increase 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. By building a scenario-based model, you will be able to better understand how the user interacts with the software, uncovering the major functions and features that stakeholder require of the system.

What is the work product? Scenario-based modeling produces a text-oriented representation call a "use case." The use case describes a specific interaction in a manner that can be informal (a simple narrative) or more structured and formal in nature. The use case can be supplemented with a number of different UML diagrams that overlay a more procedural view of the interaction.

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 earlier editions of this book, the term *analysis model* was used, rather than *requirements model*. In this edition, we'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.

uml models 179
use cases 173
use case exception 177

particularly to the Target Document [software requirements specification]. Problems of size must be dealt with using an effective method of partitioning. The Victorian novel specification is out. Graphics have to be used whenever possible. We have to differentiate between logical [essential] and physical [implementation] considerations . . . At the very least, we need . . . Something to help us partition our requirements and document that partitioning before specification . . . Some means of keeping track of and evaluating interfaces . . . New tools to describe logic and policy, something better than narrative text

Although DeMarco wrote about the attributes of analysis modeling more than three decades ago, his comments still apply to modern requirements modeling methods and notation.

9.1 Requirements Analysis

Quote:

"Any one 'view' of requirements is insufficient to understand or describe the desired behavior of a complex system."

Alan M. Davis

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 8).

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

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, we focus on *scenario-based modeling*—a technique that is growing increasingly popular throughout the software engineering community. In Chapters 10 and 11 we consider class-based models and behavioral models. Over the past decade, flow and data modeling have become less commonly used, while scenario and class-based methods, supplemented with behavioral approaches and pattern-based techniques have grown in popularity.[2]

Quote:

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

Andrew Hunt and David Thomas

9.1.1 Overall Objectives and Philosophy

Throughout analysis 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?[3]

In previous chapters, we 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.[4]

KEY POINT

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

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 12 through 18) that describes the software's application architecture, user interface, and component-level structure. This relationship is illustrated in Figure 9.1.

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.

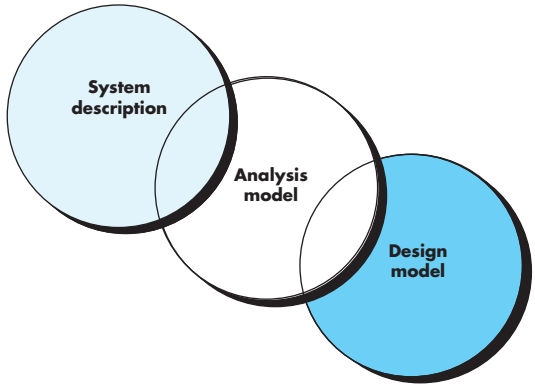
---

2 Our presentation of flow-oriented modeling and data modeling has been omitted from this edition. However, copious information about these older requirements modeling methods can be found on the Web. If you have interest, use the search phrase "structured analysis."

3 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.*

4 Alternatively, the software team may choose to create a prototype (Chapter 4) in an effort to better understand requirements for the system.

FIGURE 9.1
The requirements model as a bridge between the system description and the design model



9.1.2 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:

? Are there some basic guidelines that can guide us as we do requirements analysis work?

- *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, efforts 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 add additional diagrams when they add no new information. Don't use complex notational forms when a simple list will do.

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

9.1.3 Domain Analysis

WebRef

Many useful resources for domain analysis and many other topics can be found at **http://www.sei.cmu.edu/**.

In the discussion of requirements engineering (Chapter 8), we noted that analysis patterns often reoccur across many applications within a specific business domain. If these patterns are defined and categorized in a manner that allows 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.

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.

KEY POINT

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.

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.[5]

Using terminology that was introduced previously in this book, domain analysis may be viewed as an umbrella activity for the software process. By this we 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[6] 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.

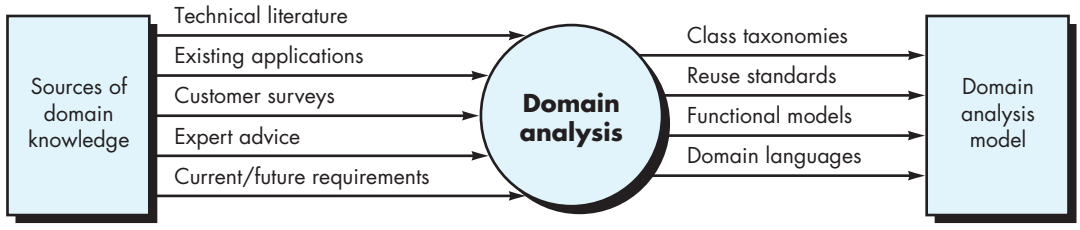
Figure 9.2 [Arn89] illustrates key inputs and outputs for the domain analysis process. Sources of domain knowledge are surveyed in an attempt to identify objects that can be reused across the domain.

---

5 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]

6 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.

FIGURE 9.2 Input and output for domain analysis



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.

9.1.4 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

Quote:

"[A]nalysis is frustrating, full of complex interpersonal relationships, indefinite, and difficult. In a word, it is fascinating. Once you're hooked, the old easy pleasures of system building are never again enough to satisfy you."

Tom DeMarco

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 4) are predominantly object oriented.

In this edition of the book, we have chosen to emphasize elements of object-oriented analysis as it is modeled using UML. Our goal is to suggest a combination of representations will provide stakeholders with the best model of software requirements and the most effective bridge to software design.

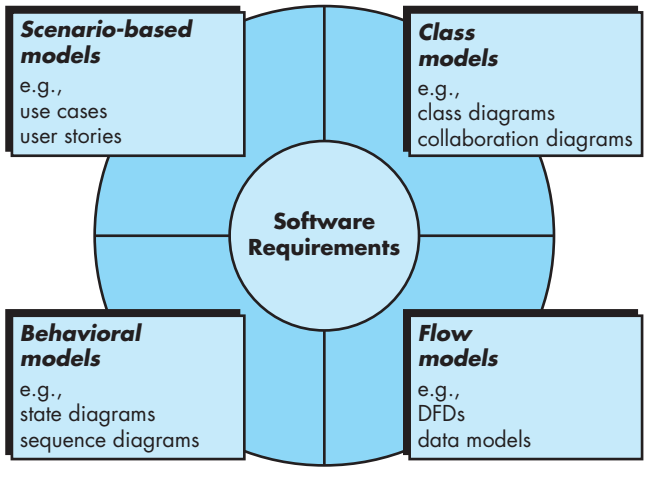
Each element of the requirements model (Figure 9.3) presents the problem from a different point of view. Scenario-based elements depict how the user interacts with the system and the specific sequence of activities that occur as the software is used. Class-based elements model the objects that the system 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.

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

Analysis modeling leads to the derivation of one or more 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.

FIGURE 9.3

Elements of the analysis model



9.2 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[7] begins with the creation of scenarios in the form of use cases, activity diagrams, and swimlane diagrams.

9.2.1 Creating a Preliminary Use Case

Quote:

"[Use cases] are simply an aid to defining what exists outside the system (actors) and what should be performed by the system (use cases)."

Ivar Jacobson

Alistair Cockburn characterizes a use case as a "contract for behavior" [Coc01b]. As we discussed in Chapter 8, the "contract" defines the way in which an actor[8] 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, we examine how use cases are developed as part of the analysis modeling activity.[9]

In Chapter 8, we 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, quality function deployment (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 9.3.1) developed as part of requirements modeling.

---

7 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.

8 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].

9 Use cases are a particularly important part of analysis modeling for user interfaces. Interface analysis and design is discussed in detail in Chapter 15.

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?

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 8 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.

Quote:

"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

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].

9.2.2 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]:

? How do I examine alternative courses of action when I develop a use case?

- *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."[10] This error condition becomes a secondary scenario.

10 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.

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.

What is a use case exception and how do I determine what exceptions are likely?

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 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).

9.2.3 Writing a Formal Use Case

The informal use cases presented in Section 9.2.1 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 9.2.2). 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: Infrequent.

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

Secondary actors: System administrator, cameras.

Channels to secondary actors:

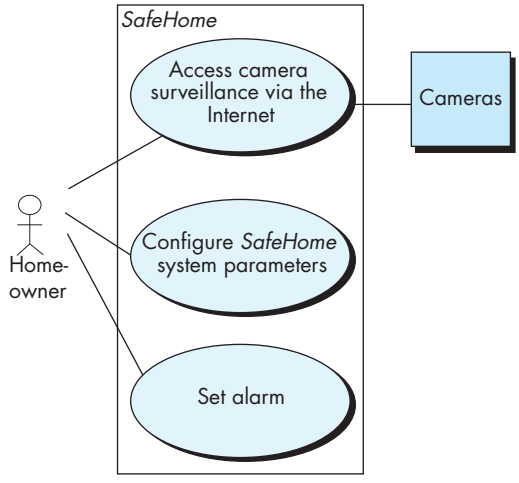
1. System administrator: PC-based system.
2. Cameras: wireless connectivity.

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?

FIGURE 9.4

Preliminary use case diagram for the *SafeHome* system



WebRef

When are you finished writing use cases? For a worthwhile discussion of this topic, see **ootips.org/use-cases-done.html**.

In many cases, there is no need to create a graphical representation of a usage scenario. However, diagrammatic representation can facilitate understanding, particularly when the scenario is complex. As we noted earlier in this book, UML does provide use case diagramming capability. Figure 9.4 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 function 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.

9.3 UML MODELS THAT SUPPLEMENT THE USE CASE

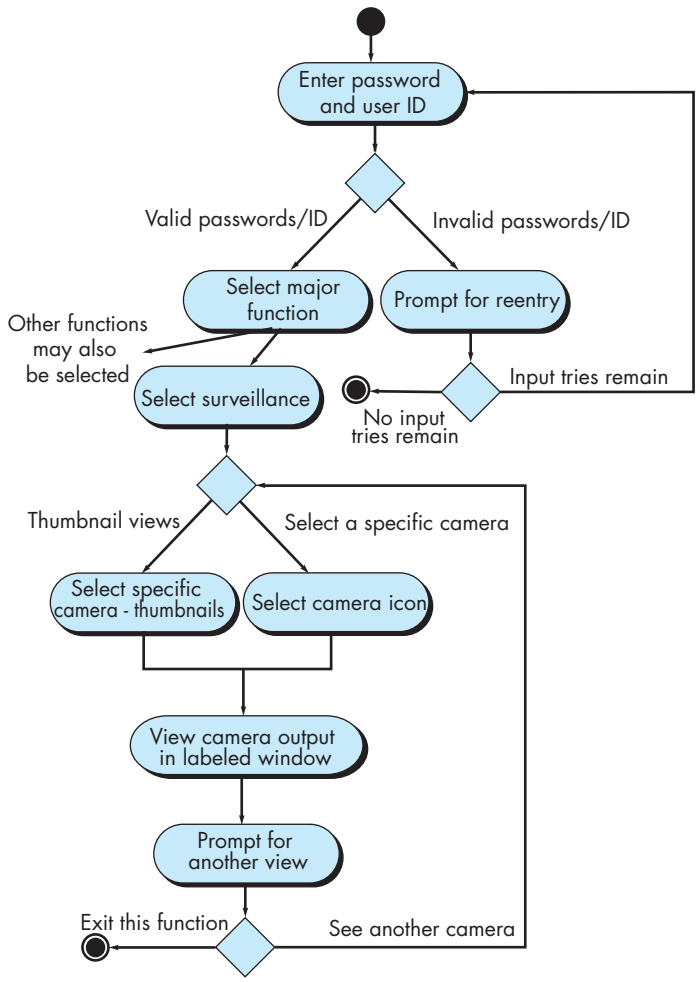
There are many requirements modeling situations in which a text-based model—even one as simple as a use case—may not impart information in a clear and concise manner. In such cases, you can choose from a broad array of UML graphical models.

9.3.1 Developing an Activity Diagram

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 9.5. 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 number of times. This is represented by a decision diamond below "Prompt for reentry."

FIGURE 9.5

Activity diagram for Access camera surveillance via the Internet—display camera views function.



9.3.2 Swimlane Diagrams

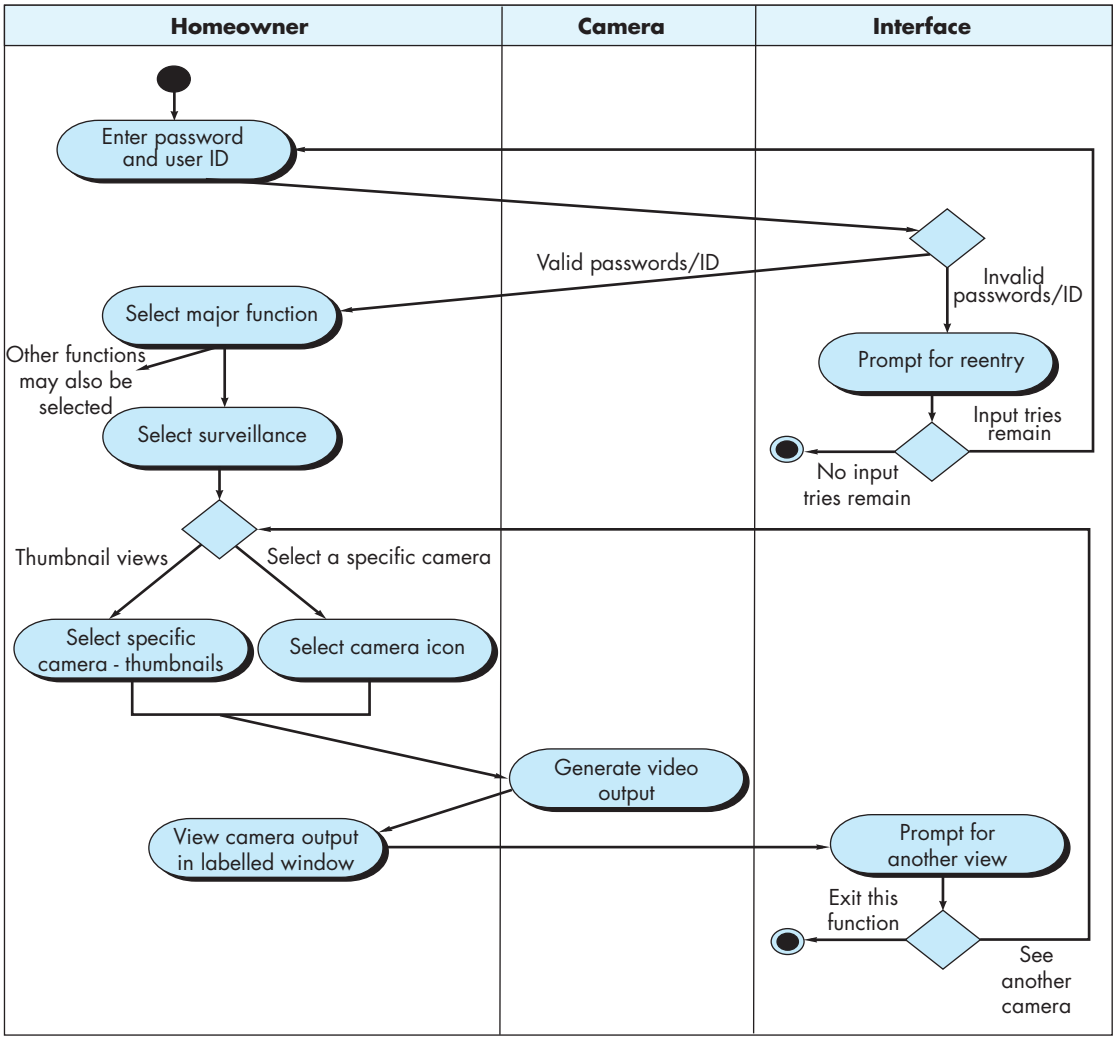
KEY POINT

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

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 (Chapter 10) has responsibility for the action described by an activity rectangle. Responsibilities are represented as parallel segments 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 9.5. Referring to Figure 9.6, the activity diagram is rearranged so that

FIGURE 9.6 **Swimlane diagram for Access camera surveillance via the Internet—display camera views function.**



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.

Quote:

"A good model guides your thinking, a bad one warps it."

Brian Marick

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 Chapters 10 and 11, we examine other dimensions of requirements modeling.

9.4 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 description 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. Swimlane diagrams illustrate how the processing flow is allocated to various actors or classes.

Problems and Points to Ponder

9.1. Is it possible to begin coding immediately after a requirements model has been created? Explain your answer and then argue the counterpoint.

9.2. 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.