REFERENCES

[AMB02] Ambler, S., and R. Jeffries, *Agile Modeling,* Wiley, 2002.

[BEN99] Bentley, J., *Programming Pearls,* 2nd ed., Addison-Wesley, 1999.

[BOE96] Boehm, B., "Anchoring the Software Process," *IEEE Software,* vol. 13, no. 4, July 1996, pp. 73–82.

[BOH00] Bohl, M., and M. Rynn, *Tools for Structured Design: An Introduction to Programming Logic,* 5th ed., Prentice-Hall, 2000.

[DAV95] Davis, A., *201 Principles of Software Development,* McGraw-Hill, 1995.

[FOW99] Fowler, M., et al., *Refactoring: Improving the Design of Existing Code,* Addison-Wesley, 1999.

[GAR95] Garlan, D., and M. Shaw, "An Introduction to Software Architecture," *Advances in Software Engineering and Knowledge Engineering,* vol. I (V. Ambriola and G. Tortora, eds.), World Scientific Publishing Company, 1995.

[HIG00] Highsmith, J., *Adaptive Software Development: An Evolutionary Approach to Managing Complex Systems,* Dorset House Publishing, 2000.

[HOO96] Hooker, D., "Seven Principles of Software Development," September 1996, available at http://c2.com/cgi/wikiSevenPrinciplesOfSoftwareDevelopment.

[HUN95] Hunt, D., A. Bailey, and B. Taylor, *The Art of Facilitation,* Perseus Book Group, 1995.

[HUN99] Hunt, A., D. Thomas, and W. Cunningham, *The Pragmatic Programmer,* Addison-Wesley, 1999.

[JUS99] Justice, T., et al., *The Facilitator's Fieldbook,* AMACOM, 1999.

[KAN93] Kaner, C., J. Falk, and H. Q. Nguyen, *Testing Computer Software,* 2nd ed., Van Nostrand-Reinhold, 1993.

[KAN96] Kaner, S., et al., *The Facilitator's Guide to Preparatory Decision Making,* New Society Publishing, 1996.

[KAR94] Karten, N., *Managing Expectations,* Dorset House, 1994.

[KER78] Kernighan, B., and P. Plauger, *The Elements of Programming Style,* 2nd ed., McGraw-Hill, 1978.

[KNU98] Knuth, D., *The Art of Computer Programming,* 3 volumes, Addison-Wesley, 1998.

[MCC93] McConnell, S., *Code Complete,* Microsoft Press, 1993.

[MCC97] McConnell, S., "Software's Ten Essentials," *IEEE Software,* vol. 14, no. 2, March/April, 1997, pp. 143–144.

[MYE78] Myers, G., *Composite Structured Design,* Van Nostrand, 1978.

[MYE79] Myers, G., *The Art of Software Testing,* Wiley, 1979.

[PAR72] Parnas, D. L., "On Criteria to Be Used in Decomposing Systems into Modules," *CACM,* vol. 14, no. 1, April 1972, pp. 221–227.

[POL45] Polya, G., *How to Solve It,* Princeton University Press, 1945.

[ROS75] Ross, D., J. Goodenough, and C. Irvine, "Software Engineering: Process, Principles and Goals," *IEEE Computer,* vol. 8, no. 5, May 1975.

[SHA95a] Shaw, M., and D. Garlan, "Formulations and Formalisms in Software Architecture," *Volume 1000—Lecture Notes in Computer Science,* Springer-Verlag, 1995.

[SHA95b] Shaw, M., et al., "Abstractions for Software Architecture and Tools to Support Them," *IEEE Trans. Software Engineering,* vol. SE-21, no. 4, April 1995, pp. 314–335.

[STE74] Stevens, W., G. Myers, and L. Constantine, "Structured Design," *IBM Systems Journal,* vol. 13, no. 2, 1974, pp. 115–139.

[TAY90] Taylor, D. A., *Object-Oriented Technology: A Manager's Guide,* Addison-Wesley, 1990.

[ULL97] Ullman, E., *Close to the Machine: Technophilia and its Discontents,* City Lights Books, 1997.

[WIR71] Wirth, N., "Program Development by Stepwise Refinement," *CACM,* vol. 14, no. 4, 1971, pp. 221–227.

[WOO95] Wood, J., and D. Silver, *Joint Application Design,* Wiley, 1995.

[ZAH90] Zahniser, R. A., "Building Software in Groups," *American Programmer,* vol. 3, nos. 7–8, July–August 1990.

Problems and Points to Ponder

5.1. Do some research of "facilitation" for the communication activity (use the references provided or others) and prepare a set of guidelines that focus solely on facilitation.

5.2. Are there other technical "essentials" that might be recommended for software engineering? State each and explain why you've included it.

5.3. Are there other management "essentials" that might be recommended for software engineering? State each and explain why you've included it.

5.4. An important communication principle states "prepare before you communicate." How should this preparation manifest itself in the early work that you do? What work products might result as a consequence of early preparation?

5.5. What three "domains" are considered during analysis modeling?

5.6. Do some research on "negotiation" for the communication activity, and prepare a set of guidelines that focus solely on negotiation.

5.7. Describe what *granularity* means in the context of a project schedule.

5.8. How does agile communication differ from tradition software engineering communication? How is it similar?

5.9. Why is it necessary to "move on"?

5.10. Why are models important in software engineering work? Are they always necessary? Are there qualifiers to your answer about necessity?

5.11. Try to summarize David Hooker's "Seven Principles for Software Development" (Section 5.1) in a brief paragraph. Try to distill his guidance into just a few sentences without using his words.

5.12. Try to add one additional principle to those stated for coding in Section 5.6.

5.13. Why is feedback important to the software team?

5.14. Do you agree or disagree with the following statement: "Since we deliver multiple increments to the customer, why should we be concerned about quality in the early increments—we can fix problems in later iterations"? Explain your answer.

5.15. What is a successful test?

Further Readings and Information Sources

Customer communication is a critically important activity in software engineering, yet few practitioner's spend any time reading about it. Books by Pardee (*To Satisfy and Delight Your Customer,* Dorset House, 1996) and Karten [KAR94] provide much insight into methods for effective customer interaction. Communication and planning concepts and principles are considered in many project management books. Useful project management offerings include: Hughs and Cotterell (*Software Project Management,* second edition, McGraw-Hill, 1999), Phillips (*The Software Project Manager's Handbook,* IEEE Computer Society Press, 1998), McConnell (*Software Project Survival Guide,* Microsoft Press, 1998), and Gilb (*Principles of Software Engineering Management,* Addison-Wesley, 1988).

Virtually every book on software engineering contains a useful discussion on concepts and principles for analysis, design and testing. Among the better offerings are books by Endres and his colleagues (*Handbook of Software and Systems Engineering,* Addison-Wesley, 2003), Sommerville (*Software Engineering,* sixth edition, Addison Wesley, 2000), Pfleeger (*Software Engineering: Theory and Practice,* Prentice-Hall, 2001) and Schach (*Object-Oriented and Classical*

Software Engineering, McGraw-Hill, 2001). An excellent collection of software engineering principles has been compiled by Davis [DAV95].

Modeling concepts and principles are considered in many books dedicated to requirements analysis and/or software design. Young (*Effective Requirements Practices,* Addison-Wesley, 2001) emphasizes a "joint team" of customers and developers who develop requirements collaboratively. Weigers (*Software Requirements,* Microsoft Press, 1999) presents many key requirements engineering and requirements management practices. Somerville and Kotonya (*Requirements Engineering: Processes and Techniques,* Wiley, 1998) discuss "elicitation" concepts and techniques and other requirements engineering principles.

Norman's (*The Design of Everyday Things,* Currency/Doubleday, 1990) is must reading for every software engineer who intends to do design work. Winograd and his colleagues (*Bringing Design to Software,* Addison-Wesley, 1996) have edited an excellent collection of essays that address practical issues for software design. Constantine and Lockwood (*Software for Use,* Addison-Wesley, 1999) present the concepts associated with "user-centered design." Tognazzini (*Tog on Software Design,* Addison-Wesley, 1995) presents a worthwhile philosophical discussion of the nature of design and the impact of decisions on quality and a team's ability to produce software that provides great value to its customer.

Hundreds of books address one or more elements of the construction activity. Kernighan and Plauger [KER78] have written a classic text on programming style, McConnell [MCC93] presents pragmatic guidelines for practical software construction, Bentley [BEN99] suggests a wide variety of programming pearls, Knuth [KNU98] has written a classic three-volume series on the art of programming, and Hunt [HUN99] suggests pragmatic programming guidelines. The testing literature has blossomed over the past decide. Myers [MYE79] remains a classic. Books by Whittaker (*How to Break Software,* Addison-Wesley, 2002), Kaner and his colleagues (*Lessons Learned in Software Testing,* Wiley, 2001), and Marick (*The Craft of Software Testing,* Prentice-Hall, 1997) each present important testing concepts and principles and much pragmatic guidance.

A wide variety of information sources on software engineering practice are available on the Internet. An up-to-date list of World Wide Web references that are relevant to software engineering practice can be found at the SEPA Web site:
http://www.mhhe.com/pressman.

CHAPTER

6 SYSTEM ENGINEERING

KEY CONCEPTS

BPE

macro elements

product engineering

system:

architecture

characteristics

elements

hierarchy

modeling

simulation

templates

UML models

Almost 500 years ago, Machiavelli said, "There is nothing more difficult to take in hand, more perilous to conduct or more uncertain in its success, than to take the lead in the introduction of a new order of things." During the past 50 years, computer-based systems have introduced a new order. Although technology has made great strides since Machiavelli spoke, his words continue to ring true.

Software engineering occurs as a consequence of a process called *system engineering*. Instead of concentrating solely on software, system engineering focuses on a variety of elements, analyzing, designing, and organizing those elements into a system that can be a product, a service, or a technology for the transformation of information or control.

The system engineering process takes on different forms depending on the application domain in which it is applied. *Business process engineering* is conducted when the context of the work focuses on a business enterprise. When a product (in this context, a product includes everything from a wireless telephone to an air traffic control system) is to be built, the process is called *product engineering.*

Both business process engineering and product engineering attempt to bring order to the development of computer-based systems. Although each is applied in a different application domain, both strive to put software into context. That is,

QUICK LOOK

What is it? Before software can be engineered, the "system" in which it resides must be understood. To accomplish this, the overall objective of the system must be determined; the role of hardware, software, people, database, procedures, and other system elements must be identified; and operational requirements must be elicited, analyzed, specified, modeled, validated, and managed. These activities are the foundation of system engineering.

Who does it? A system engineer works to understand system requirements by working with the customer, future users, and other stakeholders.

Why is it important? There's an old saying: "You can't see the forest for the trees." In this context, the "forest" is the system, and the trees are the technology elements (including software) that are required to realize the system. If you rush to build technology elements before you understand the system, you'll undoubtedly make mistakes that will disappoint your customer. Before you worry about the trees, understand the forest.

What are the steps? Objectives and more detailed operational requirements are identified by eliciting information from the customer; requirements are analyzed to assess their clarity, completeness, and consistency; a specification, often incorporating a system model, is created and then validated by both practitioners and customers. Finally, system requirements are managed to ensure that changes are properly controlled.

What is the work product? An effective representation of the system must be produced as a consequence of system engineering. This can be a prototype, a specification or even a symbolic model, but it must communicate the operational, functional, and behavioral characteristics of the system to be built and provide insight into the system architecture.

How do I ensure that I've done it right? Review all system engineering work products for clarity, completeness, and consistency. As important, expect changes to the system requirements and manage them using solid change management (Chapter 27) methods.

both business process engineering and product engineering[1] work to allocate a role for computer software and, at the same time, to establish the links that tie software to other elements of a computer-based system.

In this chapter, we focus on the management issues and the process-specific activities that enable a software organization to ensure that it does the right things at the right time in the right way.

6.1 Computer-Based Systems

The word *system* is possibly the most overused and abused term in the technical lexicon. We speak of political systems and educational systems, of avionics systems and manufacturing systems, of banking systems and subway systems. The word tells us little. We use the adjective describing *system* to understand the context in which the word is used. *Webster's Dictionary* defines system in the following way:

> 1. a set or arrangement of things so related as to form a unity or organic whole; 2. a set of facts, principles, rules, etc., classified and arranged in an orderly form so as to show a logical plan linking the various parts; 3. a method or plan of classification or arrangement; 4. an established way of doing something; method; procedure . . . .

Five additional definitions are provided in the dictionary, yet no precise synonym is suggested. *System* is a special word. Borrowing from Webster's definition, we define a *computer-based system* as

> A set or arrangement of elements that are organized to accomplish some predefined goal by processing information.

The goal may be to support some business function or to develop a product that can be sold to generate business revenue. To accomplish the goal, a computer-based system makes use of a variety of system elements:

Software. Computer programs, data structures, and related work products that serve to effect the logical method, procedure, or control that is required.

1 In reality, the term *system engineering* is often used in this context. However, for the purposes of this book system engineering is generic and is used to encompass both business process engineering and product engineering.

ADVICE

Don't be lured into taking a "software-centric" view. Begin by considering all elements of a system before you concentrate on software.

Hardware. Electronic devices that provide computing capability, the interconnectivity devices (e.g., network switches, telecommunications devices) that enable the flow of data, and electromechanical devices (e.g., sensors, motors, pumps) that provide external world function.

People. Users and operators of hardware and software.

Database. A large, organized collection of information that is accessed via software and persists over time.

Documentation. Descriptive information (e.g., models, specifications, hard-copy manuals, on-line help files, Web sites) that portrays the use and/or operation of the system.

Procedures. The steps that define the specific use of each system element or the procedural context in which the system resides.

These elements combine in a variety of ways to transform information. For example, a marketing department transforms raw sales data into a profile of the typical purchaser of a product; a robot transforms a command file containing specific instructions into a set of control signals that cause some specific physical action. Creating an information system to assist the marketing department and control software to support the robot both require system engineering.

"Never trust a computer you can't throw out a window."

Steve Wozniak

KEY POINT

Complex systems are actually a hierarchy of macro elements that are themselves systems.

One complicating characteristic of computer-based systems is that the elements constituting one system may also represent one macro element of a still larger system. The *macro element* is a computer-based system that is one part of a larger computer-based system. As an example, we consider a *factory automation system* that is essentially a hierarchy of systems. At the lowest level of the hierarchy we have a numerical control machine, robots, and data entry devices. Each is a computer-based system in its own right. The elements of the numerical control machine include electronic and electromechanical hardware (e.g., processor and memory, motors, sensors), software (for communications and machine control), people (the machine operator), a database (the stored NC program), documentation, and procedures. A similar decomposition could be applied to the robot and data entry device. Each is a computer-based system.

At the next level in the hierarchy, a manufacturing cell is defined. The *manufacturing cell* is a computer-based system that may have elements of its own (e.g., computers, mechanical fixtures) and also integrates the macro elements that we have called numerical control machine, robot, and data entry device.

To summarize, the manufacturing cell and its macro elements each are composed of system elements with the generic labels: software, hardware, people, database, procedures, and documentation. In some cases, macro elements may share a generic element. For example, the robot and the NC machine both might be managed by a single operator (the people element). In other cases, generic elements are exclusive to one system.

The role of the system engineer is to define the elements for a specific computer-based system in the context of the overall hierarchy of systems (macro elements). In the sections that follow, we examine the tasks that constitute computer system engineering.

6.2 The System Engineering Hierarchy

WebRef
The International Council of System Engineering (INCOSE) provides many useful resources at **www.incose.org**

Regardless of its domain of focus, system engineering encompasses a collection of top-down and bottom-up methods to navigate the hierarchy illustrated in Figure 6.1. The system engineering process usually begins with a "world view." That is, the entire business or product domain is examined to ensure that the proper business or technology context can be established. The world view is refined to focus more fully on a specific domain of interest. Within a specific domain, the need for targeted system elements (e.g., data, software, hardware, people) is analyzed. Finally, the analysis, design, and construction of a targeted system element is initiated. At the top of the hierarchy, a very broad context is established and, at the bottom, detailed technical activities, performed by the relevant engineering discipline (e.g., hardware or software engineering), are conducted.[2]

KEY POINT
Good system engineering begins with a clear understanding of context—the world view—and then progressively narrows focus until technical detail is understood.

Stated in a slightly more formal manner, the *world view* (WV) is composed of a set of domains (D_i), which can each be a system or system of systems in its own right.

$$WV = \{D_1, D_2, D_3, \ldots, D_n\}$$

Each domain is composed of specific *elements* (E_j) each of which serves some role in accomplishing the objective and goals of the domain or component:

$$D_i = \{E_1, E_2, E_3, \ldots, E_m\}$$

Finally, each element is implemented by specifying the technical *components* (C_k) that achieve the necessary function for an element:

$$E_j = \{C_1, C_2, C_3, \ldots, C_k\}$$

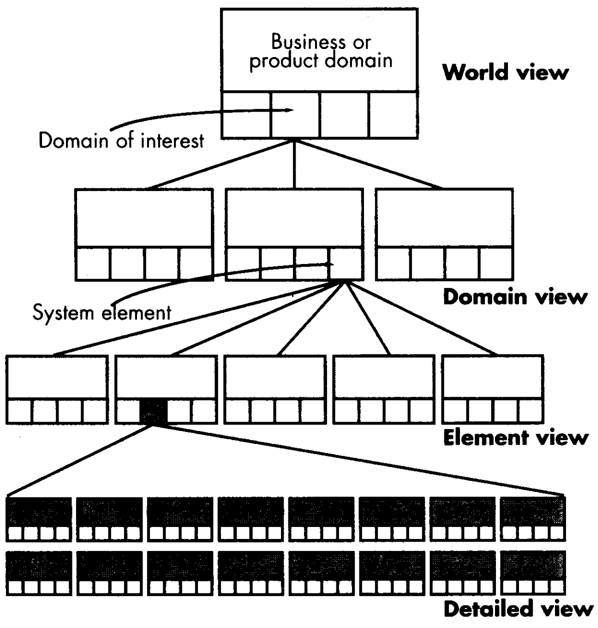
In the software context, a component could be a computer program, a reusable program component, a module, a class or object, or even a programming language statement.

"Always design a thing by considering it in its next larger context—a chair in a room, a room in a house, a house in an environment, an environment in a city plan."
Eliel Saarinen

2 In some situations, however, system engineers must first consider individual system elements. Using this approach, subsystems are described bottom-up by first considering constituent detailed components of the subsystem.

FIGURE 6.1

The system engineering hierarchy



It is important to note that the system engineer narrows the focus of work as she moves downward in the hierarchy just described. However, the world view portrays a clear definition of overall functionality that will enable the engineer to understand the domain, and ultimately the system or product, in the proper context.

6.2.1 System Modeling

System modeling is an important element of the system engineering process. Whether the focus is on the world view or the detailed view, the engineer creates models that [MOT92]:

What does a system engineering model accomplish?

- Define the processes that serve the needs of the view under consideration.
- Represent the behavior of the processes and the assumptions on which the behavior is based.
- Explicitly define both exogenous and endogenous input[3] to the model.
- Represent all linkages (including output) that will enable the engineer to better understand the view.

3 *Exogenous* inputs link one constituent of a given view with other constituents at the same level or other levels; *endogenous* input links individual components of a constituent at a particular view.

To construct a system model, the engineer should consider a number of restraining factors:

1. *Assumptions* that reduce the number of possible permutations and variations, thus enabling a model to reflect the problem in a reasonable manner. As an example, consider a three-dimensional rendering product used by the entertainment industry to create realistic animation. One domain of the product enables the representation of 3D human forms. Input to this domain encompasses the ability to specify movement from a live human actor, from video, or by the creation of graphical models. The system engineer makes certain assumptions about the range of allowable human movement (e.g., legs cannot be wrapped around the torso) so that the range of inputs and processing can be limited.

KEY POINT

A system engineer considers the following factors when determining alternative solutions: assumptions, simplifications, limitations, constraints, and customer preferences.

2. *Simplifications* that enable the model to be created in a timely manner. To illustrate, consider an office products company that sells and services a broad range of copiers, scanners, and related equipment. The system engineer is modeling the needs of the service organization and is working to understand the flow of information that spawns a service order. Although a service order can be derived from many origins, the engineer categorizes only two sources: internal demand and external request. This enables a simplified partitioning of input that is required to generate the service order.
3. *Limitations* that help to bound the system. For example, an aircraft avionics system is being modeled for a next generation aircraft. Since the aircraft has a two-engine design, the monitoring domain for propulsion will be modeled to accommodate a maximum of two engines and associated redundant systems.
4. *Constraints* that will guide the manner in which the model is created and the approach taken when the model is implemented. For example, the technology infrastructure for the three-dimensional rendering system described previously uses dual G5-based processors. The computational complexity of problems must be constrained to fit within the processing bounds imposed by these processors.
5. *Preferences* that indicate the preferred architecture for all data, functions, and technology. The preferred solution sometimes comes into conflict with other restraining factors. Yet, customer satisfaction is often predicated on the degree to which the preferred approach is realized.

The resultant system model (at any view) may call for a completely automated solution, a semiautomated solution, or a nonautomated approach. In fact, it is often possible to characterize models of each type that serve as alternative solutions to the problem at hand. In essence, the system engineer simply modifies the relative influence of different system elements (people, hardware, software) to derive models of each type.

"Simple things should be simple. Complex things should be possible."
Alan Kay

ADVICE

If simulation capability is unavailable for a reactive system, project risk increases. Consider using an incremental process model that will enable you to deliver a working product in the first iteration and then use other iterations to tune performance.

6.2.2 System Simulation

Many computer-based systems interact with the real world in a reactive fashion. That is, real-world events are monitored by the hardware and software that form the computer-based system, and based on these events, the system imposes control on the machines, processes, and even people who cause the events to occur. Real-time and embedded systems often fall into the reactive systems category.

Many systems in the reactive category control machines and/or processes (e.g., commercial aircraft or petroleum refineries) that must operate with an extremely high degree of reliability. If the system fails, significant economic or human loss could occur. For this reason, system modeling and simulation tools are used to help eliminate surprises when reactive, computer-based systems are built. These tools are applied during the system engineering process, while the role of hardware and software, databases, and people is being specified. Modeling and simulation tools enable a system engineer to "test drive" a specification of the system.

SOFTWARE TOOLS

System Simulation Tools

Objective: System simulation tools provide the software engineer with the ability to predict the behavior of a real-time system prior to the time that it is built. In addition, these tools enable the software engineer to develop mock-ups of the real-time system, allowing the customer to gain insight into the function, operation, and response prior to actual implementation.

Mechanics: Tools in this category allow a team to define the elements of a computer-based system and then execute a variety of simulations to better understand the operating characteristics and overall performance of the system. Two broad categories of system simulation tools exist: (1) general purpose tools that can model virtually any computer-based system, and (2) special purpose tools that are designed to address a specific application domain (e.g., aircraft avionics systems, manufacturing systems, electronic-systems).

Representative Tools[4]

CSIM, developed by Lockheed Martin Advanced Technology Labs (www.atl.external.lmco.com), is a general purpose discrete-event simulator for block diagram-oriented systems.

Simics, developed by Virtutech (www.virtutech.com), is a system simulation platform that can model and analyze both hardware and software-based systems.

SLX, developed by Wolverine Software (www.wolverinesoftware.com), provides general purpose building blocks for modeling the performance of a wide variety of systems.

A useful set of links to a wide array of system simulation resources can be found at http://www.idsia.ch/~andrea/simtools.html.

4 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.

6.3 BUSINESS PROCESS ENGINEERING: AN OVERVIEW

The goal of *business process engineering* (BPE) is to define architectures that will enable a business to use information effectively. When taking a world view of a company's information technology needs, there is little doubt that system engineering is required. Not only is the specification of the appropriate computing architecture required, but the software architecture that populates the organization's unique configuration of computing resources must be developed. Business process engineering is one approach for creating an overall plan for implementing the computing architecture [SPE93].

What architectures are defined and developed as part of BPE?

Three different architectures must be analyzed and designed within the context of business objectives and goals:

- Data architecture
- Applications architecture
- Technology infrastructure

The *data architecture* provides a framework for the information needs of a business or business function. The individual building blocks of the architecture are the data objects that are used by the business. A data object contains a set of attributes that define some aspect, quality, characteristic, or descriptor of the data that are being described.

Once a set of data objects is defined, their relationships are identified. A *relationship* indicates how objects are connected to one another. As an example, consider the objects: **customer** and **productA.** The two objects can be connected by the relationship *purchases;* that is, a **customer** *purchases* **productA** or **productA** *is purchased by* **customer.** The data objects (there may be hundreds or even thousands for a major business activity) flow between business functions, are organized within a database, and are transformed to provide information that serves the needs of the business.

The *application architecture* encompasses those elements of a system that transform objects within the data architecture for some business purpose. In the context of this book, we consider the application architecture to be the system of programs (software) that performs this transformation. However, in a broader context, the application architecture might incorporate the role of people (who are information transformers and users) and business procedures that have not been automated.

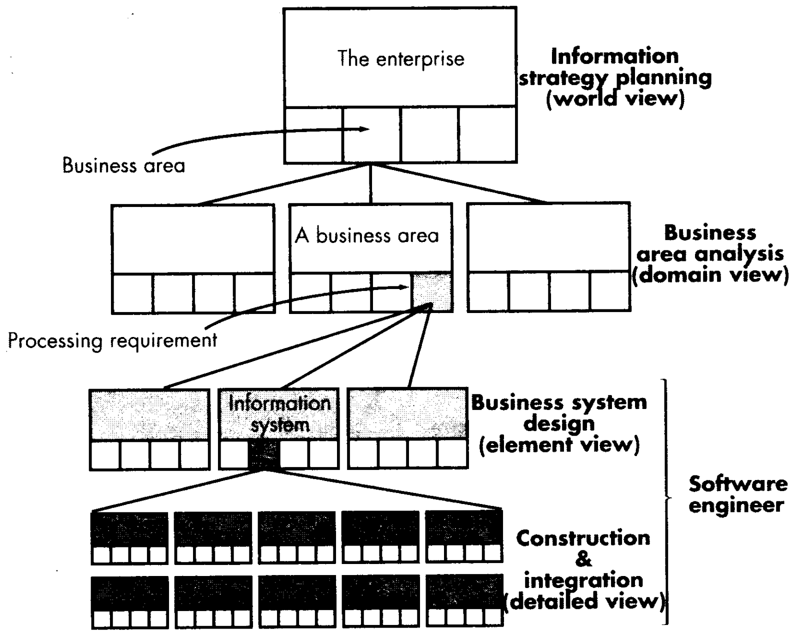
ADVICE

As a software engineer, you may never get involved in ISP or BAA. However, if it's clear that these activities haven't been done, inform stakeholders that project risk is very high.

The *technology infrastructure* provides the foundation for the data and application architectures. The infrastructure encompasses the hardware and software that are used to support the applications and data. This includes computers, operating systems, networks, telecommunication links, storage technologies, and the architecture (e.g., client/server) that has been designed to implement these technologies.

FIGURE 6.2

The business process engineering hierarchy [MAR90]


The enterprise
Information strategy planning (world view)
Business area
A business area
Business area analysis (domain view)
Processing requirement
Information system
Business system design (element view)
Software engineer
Construction & integration (detailed view)

To model these system architectures, a hierarchy of business process engineering activities is defined and illustrated in Figure 6.2.

6.4 PRODUCT ENGINEERING: AN OVERVIEW

The goal of *product engineering* is to translate the customer's desire for a set of defined capabilities into a working product. To achieve this goal, product engineering—like business process engineering—must derive architecture and infrastructure. The architecture encompasses four distinct system components: software, hardware, data (and databases), and people. A support infrastructure is established and includes the technology required to tie the components together and the information (e.g., documents, CD-ROM, video) that is used to support the components.

ADVICE

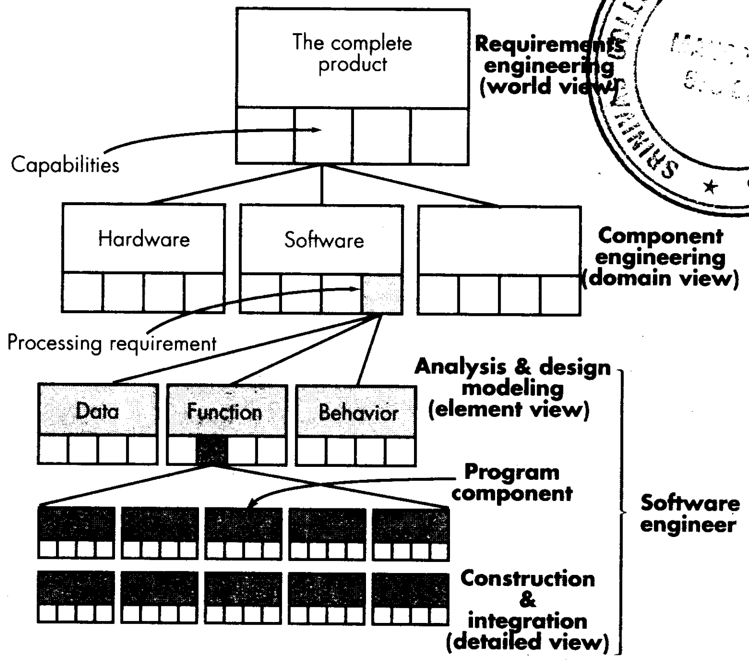
The concurrent process model (Chapter 3) is often used in this context. Each engineering discipline works in parallel. Be certain that communication is encouraged as each discipline performs its work.

Referring to Figure 6.3, the world view is achieved through requirements engineering (Chapter 7). The overall requirements of the product are elicited from the customer. These requirements encompass information and control needs, product function and behavior, overall product performance, design and interfacing constraints, and other special needs. Once these requirements are known, the job of requirements engineering is to allocate function and behavior to each of the four components noted earlier.

Once allocation has occurred, system component engineering commences. System component engineering is actually a set of concurrent activities that address each of the system components separately: software engineering, hardware engineering, human engineering, and database engineering. Each of these engineering

FIGURE 6.3

The product engineering hierarchy


The complete product
Requirements engineering (world view)
Capabilities
Hardware
Software
Component engineering (domain view)
Processing requirement
Data
Function
Behavior
Analysis & design modeling (element view)
Program component
Software engineer
Construction & integration (detailed view)

disciplines takes a domain-specific view, but it is important to note that the engineering disciplines must establish and maintain active communication with one another. Part of the role of requirements engineering is to establish the interfacing mechanisms that will enable this to happen.

The element view for product engineering is the engineering discipline itself applied to an allocated component. For software engineering, this means analysis and design modeling activities (covered in detail in later chapters) and construction and deployment activities that encompass code generation, testing, and support tasks. The analysis task models allocated requirements into representations of data, function, and behavior. Design maps the analysis model into data, architectural, interface, and software component-level designs.

SAFEHOME

Preliminary System Engineering

The scene: Software engineering team workspace after the *SafeHome* kickoff meeting has occurred.

The players: Jamie Lazar, software team member; Vinod Raman, software team member; Ed Robbins, software team member.

The conversation:

Ed: I think it went pretty well.

Vinod: Yeah . . . but all we did was look at the overall system—we've got plenty of requirements gathering work left to do for the software.

[illegible] have additional meetings
[illegible] days. By the way, I suggested
[illegible] move over here for the next
[illegible] with us so we can really
[illegible]
[illegible] go!
[illegible] looked at me like I was crazy, but
[illegible] manager] likes the
[illegible] talking to them.
[illegible] using my PDA during the meeting,
[illegible] list of basic functions.

[illegible]
[illegible] both of you a copy. Take a

[illegible]

[illegible]
[illegible] the following from Ed)
[illegible] structure/functionality of

[illegible]

[illegible] or more PCs, various
[illegible] control panels, various
[illegible] device controllers.
[illegible] wireless protocols (e.g.,
[illegible] for new-home
[illegible] within existing homes.
[illegible] of our new wireless

[illegible] that I could glean from our kick-

[illegible]

[illegible] motion sensor monitoring for
[illegible]
[illegible] and CO levels.
[illegible] levels in basement (e.g., flood or
[illegible]
[illegible] movement.
[illegible] via the Internet.

Home surveillance functions:

- Connect to one or more video cameras [illegible] inside/outside house.
- Control pan/zoom for cameras.
- Define camera monitoring zones.
- Display camera views on PC.
- Access camera views via the Internet.
- Selectively record camera output digitally.
- Replay camera output.

Home management functions:

- Control lighting.
- Control appliances.
- Control HVAC.
- Control video/audio equipment [illegible]
- Ability to set house for "vacation" [illegible] one button sets.
  - Set appliances/lighting/HVAC [illegible]
  - Set answering machine message [illegible]
  - Contacts vendors to stop [illegible]

Communication management functions:

- Answering machine functions.
  - List of callers via caller ID [illegible]
  - Messages, time-stamped [illegible]
  - Message text via voice recognition [illegible]
- E-mail functions (all standard e-mail [illegible]
  - Standard e-mail display.
  - Voice read of e-mail via phone [illegible]
- Personal phone book.
- Link to PDA.

Other functions:
As yet undefined.
All functions are accessible via the Internet [illegible] priate password protection.

6.5 System Modeling

Because a system can be represented at different levels of abstraction (e.g., the world view, the domain view, the element view), *system models* tend to be hierarchical or layered in nature. At the top of the hierarchy, a model of the complete system is presented (the world view). Major data objects, processing functions, and behaviors are represented without regard to the system component that will implement the elements of the world view model. As the hierarchy is refined or layered, component-level detail (in this case, representations of hardware, software, and so on) is modeled. Finally system models evolve into engineering models (which are further refined) that are specific to the appropriate engineering discipline.

6.5.1 Hatley-Pirbhai Modeling

Every computer-based system can be modeled as an information transform using an input-processing-output template. Hatley and Pirbhai [HAT87] have extended this view to include two additional system features—user interface processing and maintenance and self-test processing. Although these additional features are not present for every computer-based system, they are very common, and their specification makes any system model more robust.

KEY POINT The Hatley-Pirbhai model depicts input, processing, and output along with the user interface and maintenance/self-test.

Using a representation of input, processing, output, user interface processing, and self-test processing, a system engineer can create a model of system components that sets a foundation for later steps in each of the engineering disciplines.

To develop the system model, a system model template [HAT87] is used. The system engineer allocates system elements to each of five processing regions within the template: (1) user interface, (2) input, (3) system function and control, (4) output, and (5) maintenance and self-test.

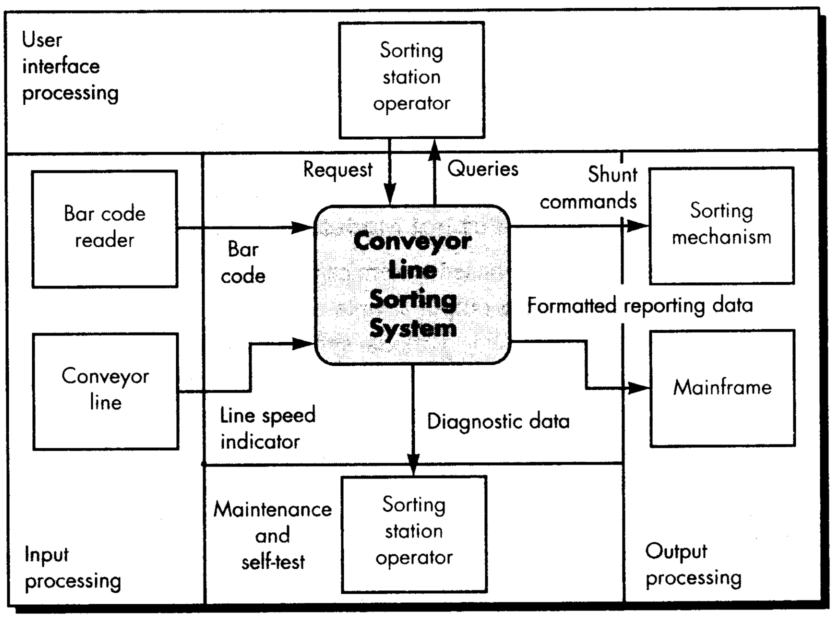
Like nearly all modeling techniques used in system and software engineering, the system model template enables the analyst to create a hierarchy of detail. A *system context diagram* (SCD) resides at the top level of the hierarchy. The context diagram "establishes the information boundary between the system being implemented and the environment in which the system is to operate" [HAT87]. That is, the SCD defines all external producers of information used by the system, all external consumers of information created by the system, and all entities that communicate through the interface or perform maintenance and self-test.

To illustrate the use of the SCD, consider a conveyor line sorting system (CLSS) described with the following (somewhat nebulous) statement of objectives:

> CLSS must be developed such that boxes moving along a conveyor line will be identified and sorted into one of six bins at the end of the line. The boxes will pass by a sorting station where they will be identified. Based on an identification number printed on the side of the box and a bar code, the boxes will be shunted into the appropriate bins. Boxes pass in random order and are evenly spaced. The line is moving slowly.
>
> A desk-top computer located at the sorting station executes all CLSS software, interacts with the bar-code reader to read part numbers on each box, interacts with the conveyor line monitoring equipment to acquire conveyor line speed, stores all part numbers sorted, interacts with a sorting station operator to produce a variety of reports and diagnostics, sends control signals to the shunting hardware to sort the boxes, and communicates with a central factory automation system.

FIGURE 6.4

System context diagram for CLSS

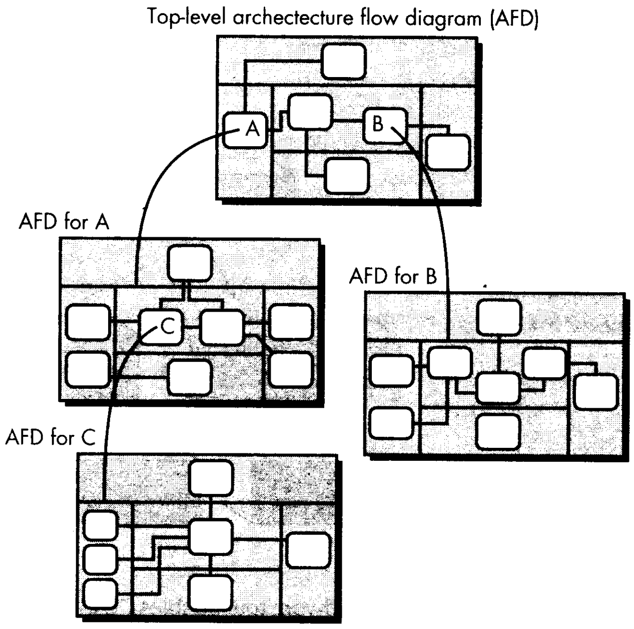


The SCD for CLSS is shown in Figure 6.4. The diagram is divided into five major segments. The top segment represents user interface processing, and the left and right segments depict input and output processing, respectively. The central segment contains process and control functions, and the bottom segment focuses on maintenance and self-test. Each box shown in the figure represents an *external entity*—that is, a producer or consumer of system information. For example, the bar-code reader produces information that is input to the CLSS system. The symbol for the entire system (or, at lower levels, major subsystems) is a rectangle with rounded corners. Hence, CLSS is represented in the processing and control region at the center of the SCD. The labeled arrows shown in the SCD represent information (data and control) as it moves from the external environment into the CLSS system. The external entity bar-code reader produces input information that is labeled bar code. In essence, the SCD places any system into the context of its external environment.

The system engineer refines the system context diagram by considering the shaded rectangle in Figure 6.4 in more detail. The major subsystems that enable the conveyor line sorting system to function within the context defined by the SCD are identified. The major subsystems are defined in a *system flow diagram* (SFD) that is derived from the SCD. Information flow across the regions of the SCD is used to guide the system engineer in developing the SFD—a more detailed "schematic" for CLSS. The system flow diagram shows major subsystems and important lines of informa-

FIGURE 6.5

Building an SFD hierarchy



tion (data and control) flow. In addition, the system template partitions the subsystem processing into each of the five regions discussed earlier. At this stage, each of the subsystems can contain one or more system elements (e.g., hardware, software, people) as allocated by the system engineer.

The initial system flow diagram becomes the top node of a hierarchy of SFDs. Each rounded rectangle in the original SFD can be expanded into another architecture template dedicated solely to it. This process is illustrated schematically in Figure 6.5. Each of the SFDs for the system can be used as a starting point for subsequent engineering steps for the subsystem that has been described.

Subsystems and the information that flows between them can be specified (bounded) for subsequent engineering work. A narrative description of each subsystem and a definition of all data that flow between subsystems become important elements of the System Specification.

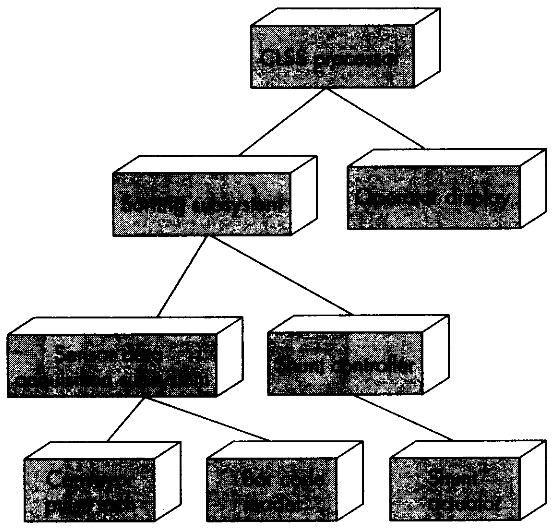
6.5.2 System Modeling with UML

UML provides a wide array of diagrams that can be used for analysis and design at both the system and the software level.[5] For the CLSS system, four important system elements

5 A more detailed discussion of UML diagrams is presented in Chapters 8 through 11. For a comprehensive discussion of UML, the interested reader should see [SCH02], [LAR01], or [BEN99].

FIGURE 6.6

Deployment diagram for CLSS hardware



are modeled: (1) the hardware that enables CLSS; (2) the software that implements database access and sorting; (3) the operator who submits various requests to the system; and (4) the database that contains relevant bar code and destination information.

CLSS hardware can be modeled at the system level using a UML *deployment diagram* as illustrated in Figure 6.6. Each 3-D box depicts a hardware element that is part of the physical architecture of the system. In some cases, hardware elements will have to be designed and built as part of the project. In many cases, however, hardware elements can be acquired off-the-shelf. The challenge for the engineering team is to properly interface the hardware elements.

WebRef
A complete specification of the syntax and semantics of the UML (discussed in later chapters) can be found at www.rational.com/uml/index.jsp.

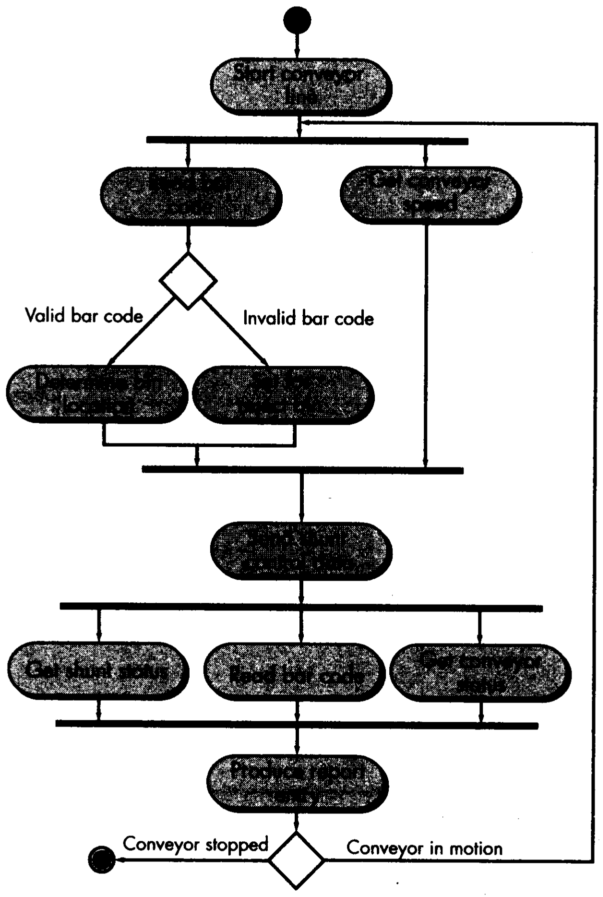
Software elements for CLSS can be depicted in a variety of ways using UML. Procedural aspects of CLSS software can be represented using an *activity diagram* (Figure 6.7). This UML notation is similar to the flowchart and is used to represent what happens as the system performs its functions. Rounded rectangles imply a specific system function; arrows imply flow through the system; the decision diamond represents a branching decision (each arrow emanating from the diamond is labeled); solid horizontal lines imply that parallel activities are occurring.

Another UML notation that can be used to model software is the *class diagram* (along with many class-related diagrams discussed later in this book). At the system engineering level, classes[6] are extracted from a statement of the problem. For

6 In earlier chapters we noted that a class represents a set of entities that is part of the system domain. These entities can be transformed or stored by the system or can serve as a producer or consumer of information produced by the system.

FIGURE 6.7

Activity diagram for CLSS



the CLSS, candidate classes might be: **Box, ConveyorLine, Bar-codeReader, ShuntController, OperatorRequest, Report, Product,** and others. Each class encapsulates a set of attributes that depict all necessary information about the class. A class description also contains a set of operations that are applied to the class in the context of the CLSS system. A UML class diagram for **Box** is shown in Figure 6.8.

The CLSS operator can be modeled with a UML use-case diagram as shown in Figure 6.9. The use-case diagram illustrates the manner in which an actor (in this case, the operator, represented by a stick figure) interacts with the system. Each labeled oval inside the box (which represents the CLSS system boundary) represents one use-case—a text scenario that describes an interaction with the system.

FIGURE 6.8

UML class diagram for Box class

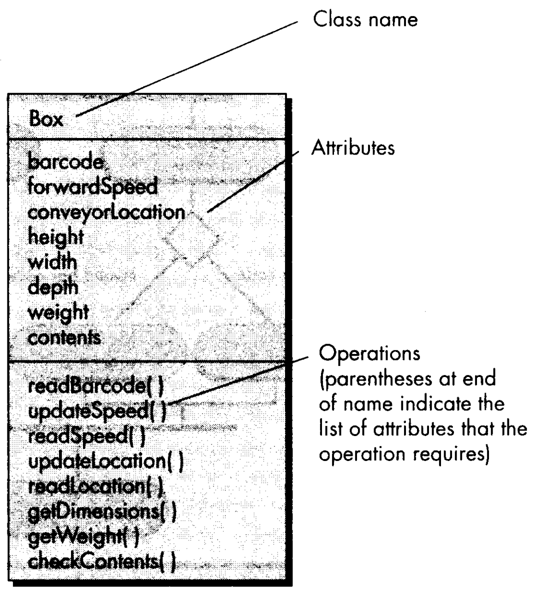
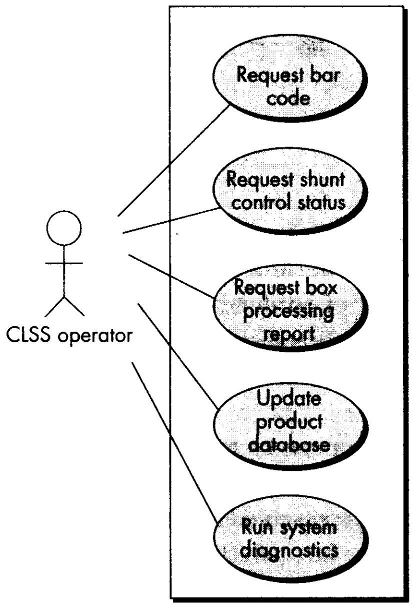


FIGURE 6.9

Use-case diagram for CLSS operator



SOFTWARE TOOLS

System Modeling Tools

Objective: System modeling tools provide the software engineer with the ability to model all elements of a computer-based system using a notation that is specific to the tool.

Mechanics: Tool mechanics vary. In general, tools in this category enable a system engineer to model (1) the structure of all functional elements of the system; (2) the static and dynamic behavior of the system; and (3) the human-machine interface.

Representative Tools[7]

Describe, developed by Embarcadero Technologies (www.embarcadero.com), is a suite of UML-based modeling tools that can represent software or complete systems.

Rational XDE and Rose, developed by Rational Technologies (www.rational.com), provide a widely used, UML-based suite of modeling and development tools for computer-based systems.

Real-Time Studio, developed by Artisan Software (www.artisansw.com), is a suite of modeling and development tools that support real-time system development.

Telelogic Tau, developed by Telelogic (www.telelogic.com), is a UML-based tool suite that supports analysis and design modeling as well as links to software construction features.

6.6 SUMMARY

A high-technology system encompasses a number of elements: software, hardware, people, database, documentation, and procedures. System engineering helps to translate a customer's needs into a model of a system that makes use of one or more of these elements.

System engineering begins by taking a "world view." A business domain or product is analyzed to establish all basic business requirements. Focus is then narrowed to a "domain view," where each of the system elements is analyzed individually. Each element is allocated to one or more engineering components, which are then addressed by the relevant engineering discipline.

Business process engineering is a system engineering approach that is used to define architectures that enable a business to use information effectively. The intent of business process engineering is to derive comprehensive data architecture, application architecture, and technology infrastructure that will meet the needs of the business strategy and the objectives and goals of each business area.

Product engineering is a system engineering approach that begins with system analysis. The system engineer identifies the customer's needs, determines economic and technical feasibility, and allocates function and performance to software, hardware, people, and databases—the key engineering components.

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.

References

[BEN99] Bennett, S., S. McRobb, and R. Farmer, *Object-Oriented Systems Analysis and Design Using UML,* McGraw-Hill, 1999.

[HAR93] Hares, J. S., *Information Engineering for the Advanced Practitioner,* Wiley, 1993, pp. 12–13.

[HAT87] Hatley, D. J., and I. A. Pirbhai, *Strategies for Real-Time System Specification,* Dorset House, 1987.

[LAR01] Larman, C., *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process,* 2nd ed., Prentice-Hall, 2001.

[MAR90] Martin, J., *Information Engineering: Book II—Planning and Analysis,* Prentice-Hall, 1990.

[MOT92] Motamarri, S., "Systems Modeling and Description," *Software Engineering Notes,* vol. 17, no. 2, April 1992, pp. 57–63.

[SCH02] Schmuller, J., *Teach Yourself UML in 24 Hours,* 2nd ed., Sams Publishing, 2002.

[SPE93] Spewak, S., *Enterprise Architecture Planning,* QED Publishing, 1993.

[THA97] Thayer, R. H., and M. Dorfman, *Software Requirements Engineering,* 2nd ed., IEEE Computer Society Press, 1997.

Problems and Points to Ponder

6.1. Select any large system or product with which you are familiar. Define the set of domains that describe the world view of the system or product. Describe the set of elements that make up one or two domains. For one element, identify the technical components that must be engineered.

6.2. Build a hierarchical "system of systems" for a system, product, or service with which you are familiar. Your hierarchy should extend down to simple system elements (hardware, software, etc.) along at least one branch of the "tree."

6.3. Although information at this point is very sketchy, try to develop one UML deployment diagram, activity diagram, class diagram, and use-case diagram for the *SafeHome* product.

6.4. Business process engineering strives to define *data* and *application architecture* as well as *technology infrastructure.* Describe what each of these terms means and provide an example.

6.5. A system engineer can come from one of three sources: the system developer, the customer, or some outside organization. Discuss the pros and cons that apply to each source. Describe an "ideal" system engineer.

6.6. Your instructor will distribute a high-level description of a computer-based system or product:

a. Develop a set of questions that you should ask as a system engineer.
b. Propose at least two different allocations for the system based on answers to your questions.
c. In class, compare your allocation to those of fellow students.

6.7. Select any large system or product with which you are familiar. State the assumptions, simplifications, limitations, constraints, and preferences that would have to be made to build an effective (and realizable) system model.

6.8. Research the literature and write a brief paper describing how modeling and simulation tools work. Alternate: Collect literature from two or more vendors that sell modeling and simulation tools and assess their similarities and differences.

6.9. Find as many single-word synonyms for the word *system* as you can. Good luck!

6.10. Are there characteristics of a system that cannot be established during system engineering activities? Describe the characteristics, if any, and explain why a consideration of them must be delayed until later engineering steps.

6.11. Develop a system context diagram for the computer-based system of your choice (or one assigned by your instructor).

6.12. Are there situations in which formal system specification can be abbreviated or eliminated entirely? Explain.

FURTHER READINGS AND INFORMATION SOURCES

Books by Hatley and his colleagues (*Process for Systems Architecture and Requirements Engineering,* Dorset House, 2000), Buede (*The Engineering Design of Systems: Models and Methods,* Wiley, 1999), Weiss and his colleagues (*Software Product-Line Engineering,* Addison-Wesley, 1999), Blanchard and Fabrycky (*System Engineering and Analysis,* third edition, Prentice-Hall, 1998), Armstrong and Sage (*Introduction to Systems Engineering,* Wiley, 1997), and Martin (*Systems Engineering Guidebook,* CRC Press, 1996) present the system engineering process (with a distinct engineering emphasis) and provide worthwhile guidance. Blanchard (*System Engineering Management,* second edition, Wiley, 1997) and Lacy (*System Engineering Management,* McGraw-Hill, 1992) discuss system engineering management issues.

Chorafas (*Enterprise Architecture and New Generation Systems,* St. Lucie Press, 2001) presents information engineering and system architectures for "next generation" IT solutions including Internet-based systems. Wallnau and his colleagues (*Building Systems from Commercial Components,* Addison-Wesley, 2001) addresses component-based systems engineering issues for information systems and products. Lozinsky (*Enterprise-Wide Software Solutions: Integration Strategies and Practices,* Addison-Wesley, 1998) addresses the use of software packages as a solution that allows a company to migrate from legacy systems to modern business processes. A worthwhile discussion of risk and system engineering is presented by Bradley (*Elimination of Risk in Systems,* Tharsis Books, 2002).

Davis (*Business Process Modeling with Aris: A Practical Guide,* Springer-Verlag, 2001), Bustard and his colleagues (*System Models for Business Process Improvement,* Artech House, 2000), and Scheer (*Business Process Engineering: Reference Models for Industrial Enterprises,* Springer-Verlag, 1998) describe business process modeling methods for enterprise-wide information systems.

Davis and Yen (*The Information System Consultant's Handbook: Systems Analysis and Design,* CRC Press, 1998) present encyclopedic coverage of system analysis and design issues in the information systems domain. An excellent IEEE tutorial by Thayer and Dorfman [THA97] discusses the interrelationship between system and software-level requirements analysis issues.

Law and his colleagues (*Simulation Modeling and Analysis,* McGraw-Hill, 1999) discuss system simulation and modeling techniques for a wide variety of application domains.

For those readers actively involved in systems work or interested in a more sophisticated treatment of the topic, Gerald Weinberg's books (*An Introduction to General System Thinking,* Wiley-Interscience, 1976 and *On the Design of Stable Systems,* Wiley-Interscience, 1979) have become classics and provide an excellent discussion of "general systems thinking" that implicitly leads to a general approach to system analysis and design. More recent books by Weinberg (*General Principles of Systems Design,* Dorset House, 1988 and *Rethinking Systems Analysis and Design,* Dorset House, 1988) continue in the tradition of his earlier work.

A wide variety of information sources on system engineering and related subjects is available on the Internet. An up-to-date list of World Wide Web references that are relevant to system engineering, information engineering, business process engineering, and product engineering can be found at the SEPA Web site:
http://www.mhhe.com/pressman.

CHAPTER

7 REQUIREMENTS ENGINEERING

KEY CONCEPTS

analysis model elements
analysis patterns
elaboration
elicitation
inception
mini-specs
negotiation
QFD
requirements management
specification
traceability
use-cases
validation

Understanding the requirements of a problem is among the most difficult tasks that face a software engineer. When you first think about it, requirements engineering 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. Requirements engineering is hard.

In the forward to a book by Ralph Young [YOU01] on effective requirements practices, I wrote:

It's your worst nightmare. A customer walks into your office, sits down, looks you straight in the eye, and says, "I know you think you understand what I said, but what you don't understand is what I said is not what I mean." Invariably, this happens late in the project, after deadline commitments have been made, reputations are on the line, and serious money is at stake.

All of us who have worked in the systems and software business for more than a few years have lived this nightmare, and yet, few of us have learned to make it go away. We struggle when we try to elicit requirements from our customers. We have trouble understanding the information that we do acquire. We often record requirements in a

QUICK LOOK

What is it? Requirements engineering helps software engineers to better understand the problem they will work to solve. It encompasses the set of tasks that lead to an understanding of what the business impact of the software will be, what the customer wants, and how end-users will interact with the software.

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

Why is it important? Designing and building an elegant computer program that solves the wrong problem serves no one's needs. That's why it is 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 the customer to define what is required, and then elaboration—where basic requirements are refined and modified. As the customer defines 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 customers' understanding of the problem coincide.

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

How do I ensure that I've done it right? Requirements engineering work products are reviewed with the customer and end-users 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.

> 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 would be dishonest to call requirements engineering the "solution" to the challenges noted above. But it does provide us with a solid approach for addressing these challenges.

7.1 A Bridge to Design and Construction

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 better understand need only after examining early iterations of the software; that things change so rapidly that requirements engineering 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 is flawed, and all can lead to a failed software project.

> **"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**

Requirements engineering, like all other software engineering activities, must be adapted to the needs of the process, the project, the product, and the people doing the work. From a software process perspective, requirements engineering (RE) is a software engineering action that begins during the communication activity and continues into the modeling activity.

In some cases, an abbreviated approach may be chosen. In others, every task defined for comprehensive requirements engineering must be performed rigorously.

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.

Overall, the software team must adapt its approach to RE. But adaptation does not mean abandonment. It is essential that the software team make a real effort to understand the requirements of a problem *before* the team attempts to solve the problem.

KEY POINT

Requirements engineering establishes a solid base for design and construction. Without it, the resulting software has a high probability of not meeting customers' 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, end-users), where business need is defined, user scenarios are described, functions and features are delineated, and project constraints are identified. Others might suggest that it begins with a broader system definition, where software is but one component (Chapter 6) of the larger system domain. But regardless of the starting point, the journey across the bridge takes us high above the project, allowing the software team 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.

7.2 REQUIREMENTS ENGINEERING TASKS

Requirements engineering provides the appropriate mechanism for understanding what the customer wants, analyzing need, assessing feasibility, negotiating a reasonable solution, specifying the solution unambiguously, validating the specification, and managing the requirements as they are transformed into an operational system [THA97]. The requirements engineering process is accomplished through the execution of seven distinct functions: *inception, elicitation, elaboration, negotiation, specification, validation,* and *management.*

ADVICE

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

It is important to note that some of these requirements engineering functions occur in parallel and all are adapted to the needs of the project. All strive to define what the customer wants, and all serve to establish a solid foundation for the design and construction of what the customer gets.

7.2.1 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.

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

Capers Jones

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 (a likely outcome), but it is sufficient to precipitate discussions with the software engineering organization.[2]

At project *inception,*[3] software engineers ask a set of context-free questions discussed in Section 7.3.4. The intent is to establish a basic understanding of the problem, the people who want a solution, the nature of the solution that is desired, and the effectiveness of preliminary communication and collaboration between the customer and the developer.

7.2.2 Elicitation

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

Christel and Kang [CRI92] identify a number of problems that help us understand why requirements *elicitation* is difficult:

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

- **Problems of scope.** The boundary of the system is ill-defined or the customers/users specify unnecessary technical detail that may confuse, rather than clarify, overall system objectives.
- **Problems of understanding.** The customers/users are not completely sure of what is needed, have a poor understanding of the capabilities and limitations of their computing environment, don't have a full understanding of the problem domain, have trouble communicating needs to the system engineer, omit information that is believed to be "obvious," specify requirements that conflict with the needs of other customers/users, or specify requirements that are ambiguous or untestable.
- **Problems of volatility.** The requirements change over time.

To help overcome these problems, requirements engineers must approach the requirements gathering activity in an organized manner.

7.2.3 Elaboration

The information obtained from the customer during inception and elicitation is expanded and refined during *elaboration.* This requirements engineering activity focuses on developing a refined technical model of software functions, features, and constraints.

2 If a computer-based system is to be developed, discussions begin with system engineering, an activity that defines the world-view and domain view (Chapter 6) for the system.

3 Readers of Chapter 3 will recall that the Unified Process defines a more comprehensive "inception phase" that encompasses the inception, elicitation, and elaboration tasks discussed in this chapter.

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 is an analysis modeling action (Chapter 8) that is composed of a number of modeling and refinement tasks. 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 UML diagrams are produced.

The end-result of elaboration is an analysis model that defines the informational, functional, and behavioral domain of the problem.

INFO

Analysis Modeling

Assume for a moment that you have been asked to specify all requirements for the construction of a gourmet kitchen. You know the dimensions of the room, the location of doors and windows, and the available wall space.

In order to fully specify what is to be built, you might list all cabinets and appliances (their manufacturer, model number, dimensions). You would then specify the countertops (laminate, granite, etc.), plumbing fixtures, flooring, and the like. These lists would provide a useful specification, but they do not provide a complete model of what you want. To complete the model, you might create a three-dimensional rendering that shows the position of the cabinets and appliances and their relationship to one another. From the model, it would be relatively easy to assess the efficiency of workflow (a requirement for all kitchens), and the aesthetic "look" of the room (a personal, but very important requirement).

We build analysis models for much the same reason that we would develop a blueprint or 3D rendering for the kitchen. It is important to evaluate the system's components in relationship to one another, to determine how requirements fit into this picture, and to assess the "aesthetics" of the system as it has been conceived.

7.2.4 Negotiation

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

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.

The requirements engineer must reconcile these conflicts through a process of *negotiation*. Customers, users, and other stakeholders are asked to rank requirements and then discuss conflicts in priority. Risks associated with each requirement are identified and analyzed (see Chapter 25 for details). Rough "guestimates" of development effort are made and used to assess the impact of each requirement on project cost and delivery time. Using an iterative approach, requirements are eliminated, combined, and/or modified so that each party achieves some measure of satisfaction.

4 The terms *operations* and *methods* are also used.

7.2.5 Specification

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

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.

The specification is the final work product produced by the requirements engineer. It serves as the foundation for subsequent software engineering activities. It describes the function and performance of a computer-based system and the constraints that will govern its development.

7.2.6 Validation

ADVICE

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

The work products produced as a consequence of requirements engineering are assessed for quality during a *validation* step. Requirements validation examines the specification to ensure that all software requirements have been stated unambiguously; that inconsistencies, omissions, and errors have been detected and corrected; and that the work products conform to the standards established for the process, the project, and the product.

The primary requirements validation mechanism is the formal technical review (Chapter 26). The review team that validates requirements includes software engineers, customers, users, and other stakeholders who examine the specification looking for errors in content or interpretation, areas where clarification may be required, missing information, inconsistencies (a major problem when large products or systems are engineered), conflicting requirements, or unrealistic (unachievable) requirements.

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?

7.2.7 Requirements Management

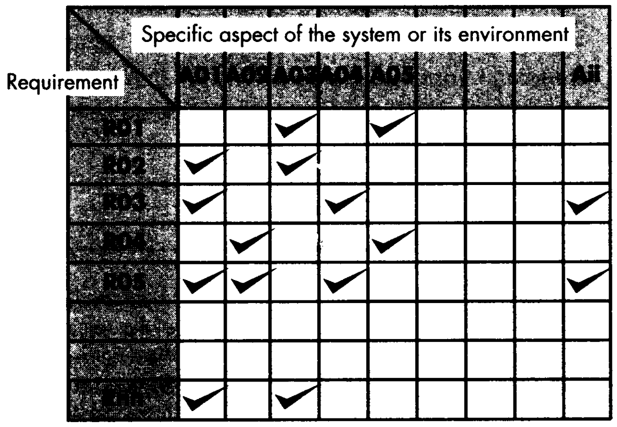
WebRef
A variety of useful information on requirements management can be obtained at www.jiludwig.com.

In Chapter 6, we noted that requirements for computer-based systems change and that 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.[5] Many of these activities are identical to the software configuration management (SCM) techniques discussed in Chapter 27.

Requirements management begins with identification. Each requirement is assigned a unique identifier. Once requirements have been identified, traceability tables are developed. Shown schematically in Figure 7.1, each *traceability table* relates requirements to one or more aspects of the system or its environment. Among many possible traceability tables are the following:

Features traceability table. Shows how requirements relate to important customer observable system/product features.

FIGURE 7.1
Generic traceability table



Requirement	Specific aspect of the system or its environment								
	A01	A02	A03	A04	A05				Aii
R01			✓		✓				
R02	✓		✓						
R03	✓			✓					✓
R04		✓			✓				
R05	✓	✓		✓					✓
Rnn	✓		✓						

5 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.

ADVICE

When a system is large and complex, determining the connections between requirements can be a daunting task. Use traceability tables to make the job a bit easier.

Source traceability table. Identifies the source of each requirement.

Dependency traceability table. Indicates how requirements are related to one another.

Subsystem traceability table. Categorizes requirements by the subsystem(s) that they govern.

Interface traceability table. Shows how requirements relate to both internal and external system interfaces.

In many cases, these traceability tables are maintained as part of a requirements database so that they can be quickly searched to understand how a change in one requirement will affect different aspects of the system to be built.

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[6]

A reasonably comprehensive (and up-to-date) listing of requirements engineering tools has been prepared by The Atlantic Systems Guide, Inc. and can be found at http://www.systemsguild.com/GuildSite/Robs/retools.html. Requirements modeling tools are discussed in Chapter 8. Tools noted below focus on requirements management.

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

OnYourMark Pro, developed by Omni-Vista (www.omni-vista.com), builds a requirements database, establishes relationships between requirements, and allows users to analyze the relationship between requirements and schedules/costs.

Rational RequisitePro, developed by Rational Software (www.rational.com), allow users to build a requirements database, represent relationships among requirements, and organize, prioritize, and trace requirements.

RTM, developed by Integrated Chipware (www.chipware.com), is a requirements description and traceability tool that also supports certain aspects of change control and test management.

It should be noted that many requirements management tasks can be performed using a simple spreadsheet or a small database system.

[illegible]

In an ideal setting, customers and software engineers work together on the same team.[7] 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.

6 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.

7 This approach is recommended for all projects and is an integral part of the agile software development philosophy.

Customer(s) 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 limited time to interact with the requirements engineer. None of these things are desirable, but all are fairly common, and the software team is often forced to work within the constraints imposed by this situation.

In the sections that follow, we discuss the steps required to initiate requirements engineering—to get the project started in a way that will keep it moving forward toward a successful solution.

7.3.1 Identifying the 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. Every 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, the requirements engineer should create of list of people who will contribute input as requirements are elicited (Section 7.4). The initial list will grow as stakeholders are contacted because every stakeholder will be asked: "Who else do you think I should talk to?"

7.3.2 Recognizing Multiple Viewpoints

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

"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

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. The job of the requirements engineer is to 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.

7.3.3 Working toward Collaboration

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 the 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.

7.3.4 Asking the First Questions

Earlier in this chapter, we noted that the 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, overall goals, and benefits. For example, the requirements engineer 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.

[illegible] some of the questions than all of the answers."

[illegible]

The next set of questions enables the software team 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?

[illegible] 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 7.4.

7.4 ELICITING REQUIREMENTS

The question and answer format described in Section 7.3.4 is useful at inception, but it is not an approach that has been overwhelmingly successful for more detailed elicitation of requirements. 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, elaboration, negotiation, and specification. An approach of this type is presented in the next section.

7.4.1 Collaborative Requirements Gathering

In order to encourage a collaborative, team-oriented approach to requirements gathering, a team of stakeholders and developers work together to identify the problem, propose elements of the solution, negotiate different approaches, and specify a preliminary set of solution requirements [ZAH90].[8]

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 are conducted and attended by both software engineers and customers (along with other interested 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.
- The goal is to identify the problem, propose elements of the solution, negotiate different approaches, and specify a preliminary set of solution requirements in an atmosphere that is conducive to the accomplishment of the goal.

To better understand the flow of events as they occur, we present a brief scenario that outlines the sequence of events that lead up to the requirements gathering meeting, occur during the meeting, and follow the meeting.

[illegible] a lot of time—the majority of project effort—not implementing or testing, but trying [illegible]

During inception (Section 7.3) basic questions and answers establish the scope of the problem and the overall perception of a solution. Out of these initial meetings, the stakeholders write a one- or two-page "product request." A meeting place, time, and date are selected and a facilitator is chosen. Members of the software team and other stakeholder organizations are invited to attend. The product request is distributed to all attendees before the meeting date.

8 This approach is sometimes called *facilitated application specification techniques* (FAST).

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.**

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 list 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.

As an example,[9] consider an excerpt from a premeeting document 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.

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

The requirements gathering team is composed of representatives from marketing, software and hardware engineering, and manufacturing. An outside facilitator is to be used.

ADVICE

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.

Each person develops the lists described previously. 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; an event priority scheme should be implemented).

---

9 The *SafeHome* 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.

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

Aldous Huxley

As the requirements gathering meeting begins, the first topic of discussion is the need and justification for the new product—everyone should agree that the product is justified. Once agreement has been established, each participant presents his lists for discussion. The lists can be pinned to the walls of the room using large sheets of paper, stuck to the walls using adhesive backed sheets, or written on a wall board. Alternatively, the lists may have been posted on an electronic bulletin board, at an internal Web site, or posed in a chat room environment for review prior to the meeting. Ideally, each listed entry should be capable of being manipulated separately so that lists can be combined, entries can be deleted, and additions can be made. At this stage, critique and debate are strictly prohibited.

ADVICE

Avoid the impulse to shoot down a customer's idea as "too costly" or "impractical." The idea here is to negotiate a list that is acceptable to all. To do this, you must keep an open mind.

After individual lists are presented in one topic area, a combined list is created by the group. The combined list eliminates redundant entries, adds any new ideas that come up during the discussion, but does not delete anything. After combined lists for all topic areas have been created, the facilitator coordinates discussion. The combined list is shortened, lengthened, or reworded to properly reflect the product/system to be developed. The objective is to develop a consensus list in each topic area (objects, services, constraints, and performance). The lists are then set aside for later action.

Once the consensus lists have been completed, the team is divided into smaller subteams; each works to develop *mini-specifications* for one or more entries on each of the lists.[10] Each mini-specification is an elaboration of the word or phrase contained on a list. For example, the mini-specification for the *SafeHome* object **Control Panel** might be:

> The **Control Panel** is a wall-mounted unit that is approximately 9 × 5 inches in size. The control panel has wireless connectively to sensors and a PC. User interaction occurs through a keypad containing 12 keys. A 2 × 2 inch LCD display provides user feedback. Software provides interactive prompts, echo, and similar functions.

Each subteam then presents its mini-specs to all attendees 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.

After the mini-specs are completed, each attendee makes a list of validation criteria for the product/system and presents her list to the team. A consensus list of

10 Rather than creating mini-specifications, many software teams elect to develop user scenarios called *use-cases*. These are considered in detail in Section 7.5.

validation criteria is then created. Finally, one or more participants (or outsiders) is assigned the task of writing a complete draft specification using all inputs from the meeting.

SAFEHOME

[illegible] Requirements Gathering Meeting

[illegible] A meeting room. The first [illegible] is in progress.

[illegible] software team member; [illegible] member; Ed Robbins, [illegible] software [illegible] of marketing; a [illegible] and a facilitator.

[illegible] board): So that's [illegible] for the home

[illegible] covers it from our

[illegible] that they wanted all [illegible] via the Internet? [illegible] function, no?

[illegible] . . . we'll have to [illegible] objects.

[illegible] constraints?

Jamie: It does, both [illegible]

Production rep: [illegible]

Jamie: We better [illegible] the system, disarm it, and [illegible] liability on our part.

Doug: Very true.

Marketing: But we [illegible] just be sure to stop an [illegible]

Ed: That's easier said than [illegible]

Facilitator (interrupting): [illegible] issue now. Let's note it [illegible] (Doug, serving as the [illegible] appropriate note.)

Facilitator: I have a [illegible] here.

(The group spends the [illegible] panding the details of [illegible]

7.4.2 Quality Function Deployment

POINT
QFD defines requirements in a way that maximizes customer satisfaction.

Quality function deployment (QFD) is a technique that translates the needs of the customer into technical requirements for software. QFD "concentrates on maximizing customer satisfaction from the software engineering process [ZUL92]." To accomplish this, QFD emphasizes an understanding of what is valuable to the customer and then deploys these values throughout the engineering process. QFD identifies three types of requirements [ZUL92]:

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

Expected requirements. These requirements are implicit to the product or system and may be so fundamental that the customer does not explicitly state

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!

them. Their absence will be a cause for significant dissatisfaction. Examples of expected requirements are ease of human/machine interaction, overall operational correctness and reliability, and ease of software installation.

Exciting requirements. These requirements reflect features that go beyond the customer's expectations and prove to be very satisfying when present. For example, word processing software is requested with standard features. The delivered product contains a number of page layout capabilities that are quite pleasing and unexpected.

In actuality, QFD spans the entire engineering process [PAR96]. However, many QFD concepts are applicable to the requirements elicitation activity. We present an overview of only these concepts (adapted for computer software) in the paragraphs that follow.

"[illegible] expectation fails, and most oft there where most it promises."

William [illegible]

WebRef

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

In meetings with the customer, *function deployment* is used to determine the value of each function that is required for the system. *Information deployment* identifies both the data objects and events that the system must consume and produce. These are tied to the functions. Finally, *task deployment* examines the behavior of the system or product within the context of its environment. *Value analysis* is conducted to determine the relative priority of requirements determined during each of the three deployments.

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. A variety of diagrams, matrices, and evaluation methods are then used to extract expected requirements and to attempt to derive exciting requirements [BOS91].

7.4.3 User Scenarios

As requirements are gathered, an overall vision of system functions and features begins to materialize. However, it is difficult to move into more technical software engineering activities until the software team understands 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 7.5.

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 user 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 (the points of 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 Web site 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 Web site 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 use-case scenario. Alternatively, the marketing person could have been asked to write the scenario, but this would be done outside the meeting.)

7.4.4 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:

- A statement of need and feasibility.
- A bounded statement of scope for the system or product.
- A list of customers, users, and other stakeholders who participated in requirements elicitation.

- A description of the system's technical environment.
- A list of requirements (preferably organized by function) and the domain constraints that apply to each.
- A set of usage scenarios that provide insight into the use of the system or product under different operating conditions.
- Any prototypes developed to better define requirements.

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

7.5 Developing Use-Cases

In a book that discusses how to write effective use-cases, Alistair Cockburn [COC01] 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.

KEY POINT
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.

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

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

WebRef
An excellent paper on use-cases can be downloaded from **www.rational.com/products/whitepapers/100622.jsp.**

Because requirements elicitation is an evolutionary activity, not all actors are identified during the first iteration. It is possible to identify primary actors [JAC92] during the first iteration and secondary actors as more is learned about the system. *Primary actors* interact to achieve required system function and derive the intended benefit from the system. They work directly and frequently with the software. *Secondary actors* support the system so that primary actors can do their work.

Once actors have been identified, use-cases can be developed. Jacobson [JAC92] suggests a number of questions[11] that should be answered by a use-case:

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

- Who is the primary actor(s), 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 three actors: the **homeowner** (a user), a **configuration manager** (likely the same person as **homeowner,** but playing a different role), **sensors** (devices attached to the system), and the **monitoring 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 interacts with the home security function in a number of different ways using either the alarm control panel or a PC:

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

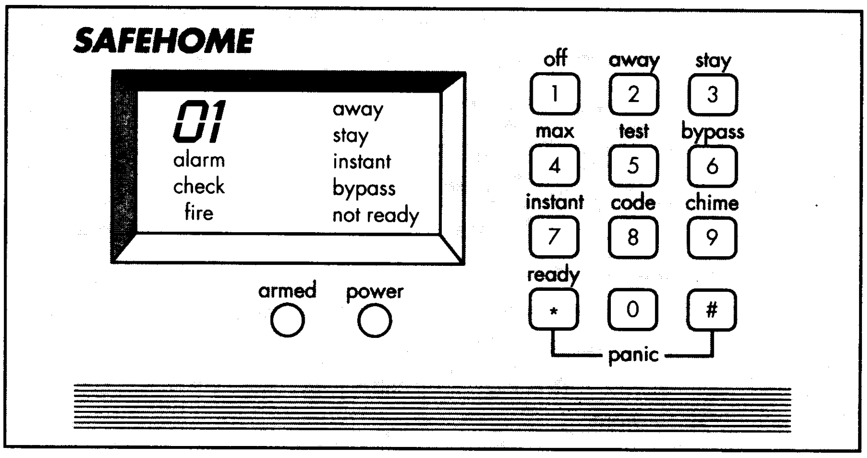
Considering the situation in which the homeowner uses the control panel, the basic use-case for system activation follows:[12]

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

12 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 is used.

FIGURE 7.2

SafeHome control panel



1. The homeowner observes the *SafeHome* control panel (Figure 7.2) 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/doors so that the *not ready* message disappears. (A *not ready* message implies that a sensor is open; i.e., that a door or window is open.)
2. The homeowner uses the keypad to key in a four-digit password. The password is compared with the valid password stored in the system. If the password is incorrect, the control panel will beep once and reset itself for additional input. If the password is correct, the control panel awaits further action.
3. The homeowner selects and keys in *stay* or *away* (see Figure 7.2) 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.

ADVICE

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

In many instances, use-case are further elaborated to provide considerably more detail about the interaction. For example, Cockburn [COC01]suggests the following template for detailed descriptions of use-cases:

Use-case:	*InitiateMonitoring*
Primary actor:	Homeowner.
Goal in context:	To set the system to monitor sensors when the homeowner leaves the house or remains inside.
Preconditions:	System has been programmed for a password and to recognize various sensors.

Trigger: The homeowner decides to "set" the system, i.e., to turn on the alarm functions.

Scenario:

1. Homeowner: observes control panel.
2. Homeowner: enters password.
3. Homeowner: selects "stay" or "away."
4. Homeowner: observes red alarm light to indicate that *SafeHome* has been armed.

Exceptions:

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

KEY POINT

Each use-case can be assessed by stakeholders, and the relative priority for each can be assigned.

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