Bahill, A. T. and Dean, F. F., Discovering system requirements, Chapter 4 in *Handbook of Systems Engineering and Management*, A. P. Sage and W. B. Rouse (Eds.), second edition, John Wiley & Sons, pp. 205-266, 2009.

4 Discovering System Requirements

A. TERRY BAHILL and FRANK F. DEAN

4.1 INTRODUCTION

A requirement is a statement that identifies a capability or function that is needed by a system in order to satisfy its customer's needs. A functional requirement defines what, how well, and under what conditions one or more inputs must be converted into one or more outputs at the boundary in question in order to satisfy the customer's needs. A customer's need might be to solve a problem, achieve an objective, or satisfy a contract, standard, or specification.

No two systems have identical requirements. However, there is an identifiable process for logically discovering the system requirements throughout the system life cycle regardless of system purpose, size, or complexity (Grady, 1993). The purpose of this chapter is to reveal this process.

This chapter explains only a part of the systems requirements process. This chapter does not discuss (1) commercial requirements management tools such as DOORS, RTM, Requisite Pro, Vital-Link, Slate, or Excel; (2) tools for modeling systems such as UML, SysML, and functional decomposition; (3) methods for flowing down requirements from the system to subsystems, from subsystems to components, and so on; or (4) scripts for producing specific reports such as a Requirements Allocation Sheet (RAS) or a Requirements Verification and Traceability Matrix (RVTM). The terminology of this chapter is a generalization of common usage at many companies. But it does not match any particular company, branch of government, or tool. For example, we use the term requirements broadly. Some people use the term only for requirements that are contained in a signed-off system specification.

A set of requirements should use a common language with a restricted vocabulary. Each project should create a special dictionary on involved computers with a restricted set of words for use by the spelling checker. Project-specific terms should be defined in a glossary. Acronyms should be expanded and put in a list.

4.2 STATING THE PROBLEM

Before we can effectively discover and develop requirements, we need a well-defined and well-stated problem. Stating the problem properly is one of the systems engineer's most important tasks. Problem stating is more important than problem solving, because an elegant solution to the wrong problem is less than worthless. The problem must be stated in a clear, unambiguous manner. The problem statement explains the customer's needs, states the goals of the project, defines the business needs, prescribes the system's capabilities, delineates the scope of the system, expresses the concept of operations, describes the stakeholders, lists the deliverables, and presents the key decisions that must be made.

State the problem in terms of needed capabilities and not in terms of preconceived solutions. A flood washed out a bridge across the Santa Cruz River near Tucson, Arizona, and made it difficult for the Indians at Mission San Xavier del Bac to get to the Bureau of Indian Affairs Health Center. A common way of stating this problem was: "We must rebuild the bridge across the Santa Cruz River." However, a better way would be to say: "The Indians at San Xavier Mission need a convenient way to get to their health center."

For examples of stating the problem without reference to specific implementations, read some U.S. patents. A patent does not protect the system; it protects the idea behind the system. Here is an excerpt from the patent on the Bat Chooser™, a system that computes the Ideal Bat Weight™ for individual baseball and softball batters (Bahill and Karnavas, 1992).

> The invention relates to an instrument for measuring a player's bat speed. Over a number of swings, the bat speed data can be used to determine the maximum ball speed after contact. The optimal bat weight is determined. Baseball and softball are examples in which the use of the instrument would be appropriate. The invention also relates to a process of using the instrument to obtain data on bat speed, plotting the bat speed data, fitting a best fit curve to the data, using the best fit curve of physiological data to obtain the ball speed after contact, plotting the ball speed, determining the maximum-ball-speed bat weight and the optimal bat weight. It is the coupling of the physiological equations to the conservation of momentum equations of Physics which is unique.

It is good engineering practice to state the problem in terms of the capabilities that the system must have or the top-level functions that the system must perform. However, it might be better to state the problem in terms of the deficiency that must be ameliorated. This stimulates consideration of more alternative designs.

Example 1

Top-Level Function. The system shall hold together 2 to 20 pieces of $8\frac{1}{2}$ by 11-inch, 20 pound paper.

Alternatives. Staple, paper clip, fold the corner, put the pages in a folder.

Example 2

The Deficiency. My reports are typically composed of 2 to 20 pieces of $8\frac{1}{2}$ by 11-inch, 20 pound paper. The pages get out of order and become mixed up with pages of other reports.

Alternatives. Staple, paper clip, fold the corner, put the pages in folders, number the pages, put them in envelopes, put them in three-ring binders, throw away the reports, convert them to electronic form, have them bound as books, put them on audio tapes, distribute them electronically, put them on floppy disks, put them on microfiche, transform the written reports into videotapes.

Do not believe the first thing your customer says. Verify the problem statement with the customer and expect to iterate this procedure several times. For an excellent (and enjoyable) reference on stating the problem, see Gause and Weinberg (1990), an excerpt of which is given in Section 4.4.

4.2.1 Define the Stakeholders

The stakeholders include all the people, organizations, and institutions that are a part of the system environment because the system provides some benefit to them and they have an interest in the system. This includes end users, operators, bill payers, owners, regulatory agencies, victims, sponsors, maintainers, architects, managers, customers, surrogate customers, testers, quality assurance, risk management, purchasing, and the environment.

Let us now illustrate some of these stakeholder roles for a commercial airliner, such as the Boeing 787. The users are the passengers who fly on the airplane. The operators are the crew who fly the plane and the mechanics who maintain it. The bill payers are the airline companies, such as United, Southwest Airlines, and so on. The owners are the stockholders of these companies. The Federal Aviation Administration (FAA) writes the regulations and certifies the airplane. Among others, people who live near the airport are victims of noise and air pollution. If the plane were tremendously successful, Airbus (the manufacturer of a competing airplane) would also be a victim. The sponsor, in this example, would be the corporate headquarters of Boeing.

However, because systems engineering delivers both a *product* and a *process* for producing it, we must also consider the stakeholders of the process. The users and operators of the process would be the employees in the manufacturing plant. The bill payer would be Boeing. The owner would be the stockholders of Boeing. The regulators would include the Occupational Safety and Health Administration (OSHA). Victims would be physically injured workers and, according to Deming, workers who have little control of the output, but who are reviewed for performance (Deming, 1982; Latzko and Saunders, 1995).

4.2.2 Identify the Audience

Before writing a document, you should identify the audience. For a requirements document, the audience is the customer and the system developer including the designers, producers, and testers.

The customer and the developer have different backgrounds and needs. Therefore, Wymore (1993) suggests two different documents for these two different groups. The *Customer Requirements Document* is a detailed description of the problem in plain language. It is intended for management, the customer, and engineering. The *Derived Requirements Document* is a succinct mathematical description or model of the requirements as described in the Customer Requirements Document. Its audience is engineering. Sometimes the derived requirements are referred to as technical, or design or product, requirements. Each derived requirement must be traceable to a customer requirement or a document, such as a vision/mission statement or a concept of operations.

Stakeholder needs, expectations, constraints, and interfaces are transformed into customer requirements. Customer requirements are nontechnical and use the customer's vocabulary. Derived requirements are the expression of these requirements in technical terms that are used for design decisions. An example of this translation is given in the first QFD chart, which maps the customer desires (whats) into technical parameters (hows) (Bahill and Chapman, 1993).

4.2.3 Avoid Using the Word Optimal

The word optimal (or optimize) should not appear in the problem statement, because there is no single optimal solution for a complex systems problem. Most system designs have many performance and cost criteria. Alternative designs satisfy these criteria to varying degrees. Moving from one alternative to another often improves satisfaction of one criterion and worsens satisfaction of another; that is, there will be trade-offs. None of the feasible alternatives is likely to optimize all the criteria. Therefore, we must settle for less than optimality.

It might be possible to optimize some subsystems, but when they are interconnected, the overall system may not be optimal. The best possible system will not be that made up of optimal subsystems. An All Star team may have optimal people at all positions, but is it likely that such an All Star team could beat the World Champions? For example, in football a Pro Bowl team is not likely to beat the Super Bowl champions.

If the requirements demanded an optimal system, data could not be provided to prove that any resulting system was indeed optimal. A substantial amount of talent, money, and time could be wasted trying to prove or demonstrate this. Instead, these resources should be used to design a better system. In general, it can be proved that a system is at a local optimum, but it cannot be proved that it is at a global optimum.

Systems engineers design satisfying systems rather than optimized systems (Simon, 1957). To promote reuse they design generalized systems rather than specialized systems. No complex system is likely to be optimal for all the people, all the time.

Humans are not optimal animals. Shrews are smaller. Elephants are bigger. Cheetahs can run faster. Porpoises can swim faster. Dolphins have bigger brains. Bats have wider bandwidth auditory systems. Deer have more sensitive olfaction systems. Butterflies are more beautiful. Pronghorn antelope have sharper vision. For color vision, raptors have four types of cones, rather than a human's three. Humans have not used evolution to optimize these systems. Humans have remained generalists. The frog's visual system has evolved much farther than that of humans: frogs have cells in the superior colliculus that are specialized to detect moving flies. Leaf cutting ants had organized agricultural societies millions of years before humans. Although humans are not optimal in any sense, they seem to rule the world.

If it is required that optimization techniques be used, then they should be applied only to subsystems and only late in the design process. However, total system performance must be analyzed to decide if the cost of optimizing a subsystem is worthwhile. Furthermore, total system performance should be analyzed over the whole range of operating environments and trade-off functions, because what is optimal in one environment with one trade-off function will probably not be optimal with others.

Because of the rapid rate at which technology is advancing, flexibility is more important than optimality. A company could buy a device, spend many person-years optimizing its inclusion into the company's system, and then discover that a new device is available that performs better than the optimized system and costs less. Besides optimal, other deprecated words include minimize, maximize, and simultaneous.

4.3 WHAT ARE REQUIREMENTS?

A requirement is a statement that identifies a capability or function needed by a system in order to satisfy a customer need. A functional requirement defines what, how well, and under what conditions one or more inputs must be converted into one or more outputs at the boundary in question in order to satisfy the customer's needs. The Capability Maturity Model Integration (CMMI, 2006) says that a requirement is (1) a condition or capability needed by a user to solve a problem or achieve an objective or (2) a condition or capability that must be possessed by a product to satisfy a contract, standard, or specification. Requirements should state what the system is to do, but they should not specify how the system is to do it. Section 4.3.1 presents an example of a requirement.

4.3.1 Example of a Requirement (Sommerville, 1989)

Graphic Editor Facility. To assist in positioning items on a diagram, the user may turn on a grid in either centimeters or inches, via an option on a control panel. Initially the grid is off. The grid may be turned on or off at any time during an editing session and can be toggled between inches and centimeters at any time. The grid option will also be provided on the reduce-to-fit view, but the number of grid lines shown will be reduced to avoid filling the diagram with grid lines.

Good Points About This Requirement. It provides rationale for the items: it explains why there should be a grid. It explains why the number of grid lines should be reduced for the reduce-to-fit view. It provides initialization information: initially the grid is off.

Bad Points. The first sentence has more than one component: (1) it states that the system should provide a grid, (2) it describes the grid units (centimeters and inches), and (3) it tells how the user will activate the grid. This requirement provides initialization information for some but not all similar items: it specifies that initially the grid is off, but it does not specify the units when it is turned on. Section 4.3.2 shows how this requirement might be improved.

4.3.2 Example of an Improved Requirement (Sommerville, 1989)

The Grid Facility

1. The graphic editor grid facility shall produce a pattern of horizontal and vertical lines forming squares of uniform size as a background to the editor window. The grid shall be passive rather than active. This means that alignment is the responsibility of the user and the system shall not automatically align items with grid lines.

   Rationale. A grid helps the user to create a neat diagram with well-spaced entries. Although an active grid might be useful, it is best to let the user decide where the items should be positioned.

2. When used in the “reduce-to-fit” mode, the logical grid line spacing shall be increased

   Rationale. If the logical grid line spacing were not increased, the background would become cluttered with grid lines.

   Specification. Eclipse/Workstation/Defs:Section 2.6.

This requirement definition refers to the requirement specification, which provides details such as units of centimeters and inches and the initialization preferences.

Simple Examples of Requirements on Requirements

Each requirement shall describe only one function.

Requirements shall be unambiguous.

Requirements shall be verifiable.

Requirements shall be prioritized.

The set of requirements shall be complete.

The set of requirements shall be consistent.

4.4 QUALITIES OF A GOOD REQUIREMENT

What distinguishes a good requirement form a bad one? The IEEE says that “requirements must be unambiguous, complete, correct, traceable, modifiable, understandable, verifiable, and ranked for importance and stability.” Martin Fowler and Michael Fagan both say that if you design your tests when you write your code, then you will have fewer defects. Therefore, each requirement should have a description of the verification procedure. These qualities and a few dozen others are presented in this section. As stated in the introduction, statements about requirements cannot be dogmatic. Each statement has been rightfully violated many times. Many other people have also written about the qualities that make a requirement good (Hooks and Farry, 2001; Young, 2001, 2006).

1. *Describes What, Not How.* There are many characteristics of a good requirement. First and foremost, a good requirement defines what a system must do, but does not specify how to do it. A statement of a requirement should not be a preconceived solution to the problem that is to be solved. To avoid this mistake, ask why the requirement is needed, then derive the real requirements. For example, it would be a mistake to require a relational database for the requirements (Hooks and Farry, 2001). The following requirements state what is needed, not how to accomplish it: “The system shall provide the ability to store requirements”. “The system shall provide the ability to sort requirements”. “The system shall provide the ability to add attributes to requirements”. It should be noted that because QFD is often used iteratively to define requirements, the *hows* in one QFD chart become the *whats* in the next, possibly making the above statements confusing. This property could also be confusing because we say *what* a system is supposed to do, and then the designers decide *how* it should do it. Then this *how* becomes a *what* at the next level down, as shown in Figure 4.1.

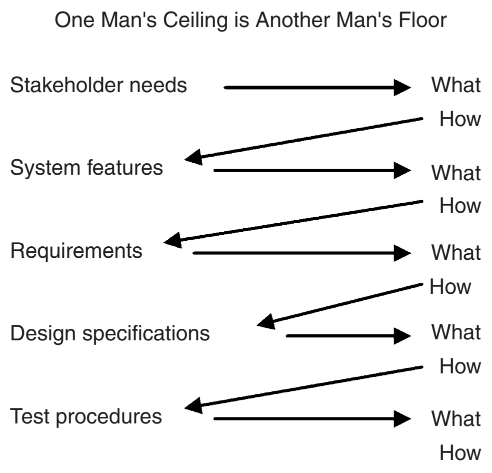

Figure 4.1 The relationship between *whats* and *hows* at different levels. (Copyright © 2004, A. T. Bahill, from http://www/sie.arizona.edu/sysengr/slides/. Used with permission.)

Requirement writers often fall into the implementation trap, when they forget that they are writing requirements and begin to think about *how* to build the system. Requirement statements should not be made confusing by descriptions of design or other implementation-specific information. Sometimes, the implementation information created by requirement writers is of value and should be preserved, but not as requirements. Nonrequirement information should be provided as working papers routed to the design team.

Frequently, requirement writers levy requirements on the function rather than on the system. For example, "The Sort function shall. . . ." This practice permits a very convenient sorting of requirements by function. This is all right. However, sometimes requirement writers mistakenly specify exchanges of information between functions as requirements. For example, "The Sort function shall notify the Store function. . . ." Such statements are not requirements, since they do not depend on external stimuli and generate no results that can be measured at the system boundary. Such language is appropriate for design documents—not specifications.

2. *Atomic (or Unitary or Single-Minded).* A requirement should be "atomic," meaning it is not easily broken into smaller requirements. There should be one concept per requirement. Complex requirements need to be decomposed into individual requirements. Requirements should be decomposed to the point where each statement defines stimuli and results associated with a single function (one or more stimuli cause a single result) for the system or component, considered as a black box. If this test cannot be satisfied, because the requirement addresses more than one function, then the requirement needs to be decomposed.

3. *Allocation.* Each requirement should be allocated to a single entity: except for nonallocated requirements (such as quality of workmanship and electrostatic discharge protection) that are spread across the whole project, like peanut butter. It is acceptable to assign two or more requirements to one physical component. However, it would most likely be a mistake to assign one requirement to two physical components (Bahill and Botta, 2008).

4. *Unique.* A requirement should have a unique label, a unique name, and unique contents. Avoid repeating requirements. However, some systems engineers say that the designers do not know what is in the requirements specification. Therefore, they deliberately repeat critical safety requirements hoping that this will increase the chances that the designers will notice them.

5. *Documented and Accessible.* A requirement must be documented (text, equations, images, databases, etc.) and the documentation must be accessible. In situations where confidentiality is important, each requirement should clearly indicate classification status. Only individuals with the appropriate clearance and the need to know should have access to classified requirements.

6. *Identifies Its Owner.* A good requirement identifies its owner. The requirement's owner must approve of any change in the requirement. This owner could be an individual or a team. The owner has certain responsibilities such as monitoring the associated technical performance measures (TPMs) or risk activities. If the owner is part of the system development team, he/she should know and communicate with the customer's representative who is responsible or most knowledgeable about the requirement or the customer's representative who generated the high-level goal or capability that led to this customer requirement. The owner is responsible for helping to define and track the trade-offs of the requirement with other requirements and for monitoring the trade-off studies that involve the requirement. The owner is also responsible for ensuring that the used requirements management tools properly interpret and track the owner's requirements.

7. *Identifies Its Target.* A good requirement identifies the target of the requirement, that is, for whom it is a requirement (e.g., the system, the process, the company, the customer). The requirement writer sometimes constructs a requirement levied on the user of the system and not on the system itself: for example, if the requirements analyst wrote "The user shall input a start time for message ingest," then someone would have to rewrite this as "The system shall enable the user to designate a start time for message ingest."

8. *Approved.* After a requirement is written, revised, reviewed, or rewritten, it must be approved by its owner. Furthermore, each top-level requirement must be approved by the customer.

9. *Traceable.* A good requirement is traceable: it should be possible to trace each requirement back to its source. See Figure 4.1. A requirement should identify related requirements (i.e., parents, children, siblings) and requirements that would be impacted by changes to it. A requirements document should have a tree (or graph) structure and this structure should be evident, often in the numbering scheme.

10. *Necessary.* Each requirement should be necessary. Systems engineers should ask: "Is this requirement really necessary?" "Can the system meet the customer's real needs without it?" If yes, then the requirement is not necessary. Avoid overspecifying the system, writing pages and pages that probably no one will ever read. There are two common types of overspecification: gold plating and specifying unnecessary things. For example, requiring that the outside of a CPU box be gold-plated is not a good requirement, because something far less expensive would probably be just as effective. Also, requiring that the inside of the CPU box be painted pink is probably an unnecessary request. Overspecification (of both types) is how $700 toilet seat covers

and $25,000 coffeepots were created (Hooks, 1994). Each requirement should include a statement of its rationale. This might help rule out unnecessary requirements.

Early in the space program, NASA contracted for a pen to be used by astronauts in space. Each pen had to write upside down, under water, over grease, and at temperatures of $-50\ ^\circ$F to $+400\ ^\circ$F. The solution had a sealed cartridge with a sliding float separating the ink from pressurized nitrogen. It cost them thousands of dollars. (Today you can buy the Fisher Space Pen for $30.) The Russian solution? Use a pencil.

11. *Complete.* Each requirement must be complete. All conditions under which the requirement applies should be stated. It is all right to have a To Be Determined (TBD) in a requirement. But each TBD should have a scheduled resolution date and should be tracked in the risk management plan. It is all right to have To Be Reviewed (TBR) in a requirement. These preliminary values are proposed with the caveat that they be reexamined at a specific review.

Each individual requirement must be complete and the requirements set must also be complete. Many times, due to the structure of the requirements set, your computer can identify incompleteness (Davis and Buchanan, 1984). Putting your requirements into a Zachman framework will also help identify incompleteness (Bahill et al., 2006).

12. *Is Not Written Negatively.* The requirement, "Do not use wire with Kapton insulation," would be very difficult to verify for commercial off-the-shelf (COTS) equipment.

13. *Unambiguous.* Can the requirement be interpreted in more than one way? If so, then the requirement should be clarified or removed. Avoid the use of synonyms (e.g., "The software requires 8 Mbytes of RAM but 12 Mbytes of memory are recommended") and homonyms (e.g., "Summaries of disk X-rays should be stored on a disk." "Time flies like an arrow and fruit flies like a banana").

14. *Is Not Always Written.* It must be noted that all systems will undoubtedly have many "commonsense" requirements that will not be written. This is acceptable as long as the requirements really are common sense. An exhaustive list of requirements would take years upon years and use reams of paper, and even then you would probably never finish.

15. *Verifiable.* Quantitative values must be given in requirements. A requirement states a necessary attribute of a system to be designed. The designer cannot design the system if a magnitude is not given for each attribute. Without quantification, system failure could occur because (1) the system exceeded the minimum necessary cost due to overdesign, or (2) it failed to account for a needed capability. Quantitative values for attributes are also necessary in order to test the product to verify that it satisfies its requirements.

Each requirement must be verifiable by test, demonstration, inspection, logical argument, analysis, modeling, or simulation. Qualitative words like *low* and *high* should be (at least roughly) defined. What is low cost to a big corporation and what is low cost to a small company may be very different. Only requirements that are clear and concise will easily be testable. Requirements with ambiguous qualifiers will probably have to be refined before testing will be possible. Furthermore, the value given should be fully described as, for example, an expected value, a median, a minimum, a maximum, or the like. A requirement such as "Reliability shall be at least 0.999" is a good requirement because it is testable, quantified, and the value is fully described as a minimum. Also, the requirement "The car's gas mileage should be about 30 miles per gallon" is a good requirement as it establishes a performance measure and an expected value. Moody et al. (1997) present a few dozen metrics that can be used to evaluate performance requirements.

The verification method and level at which the requirement can be verified should be determined explicitly as part of the development for each of the requirements. The verification level is the location in the system where the requirement is met (e.g., the "system level," the "segment level," and the "subsystem level").

Note that often the customer will state a requirement that is not quantified: for example, "The system should be aesthetically pleasing." It is then the engineer's task to define a requirement that is quantified, such as "The test for aesthetics will involve polling two hundred potential users; at least 70% shall find the system aesthetically pleasing".

It is also important to make the requirements easily testable. NASA once issued a request for proposals for a radio antenna that could withstand earthquakes and high winds. It was stated that the antenna shall not deflect by more than 0.5 degree in spite of a 0.5G force, 100 knot steady winds, or gusts of up to 150 knots. They expected bids around $15 million. But all of their bids were around $30 million. NASA asked the contractors why the bids were so high, and the contractors said testing the system was going to be very expensive. NASA revised the requirements to "When 'hit with a hammer,' the antenna shall have a resonant frequency less than 0.75 Hz". Then they got bids between $12 and $15 million (Eb Rechtin, personal communication, 1996).

16. *States Its Units of Measurement.* "People sometimes make errors," said Dr. Weiler of NASA. "The problem here was not the error, it was the failure of NASA's systems engineering.... That's why we lost the [Mars Climate Orbiter] spacecraft." One team used English units (e.g., feet and pounds) while the other used SI units (meters and kilograms) (NASA, 2000).

17. *Identifies Applicable States.* Some requirements only apply when the system is in certain states or modes. If the requirement is only to be met sometimes, the requirement statement should reflect when. There may be two requirements that are not intended to be satisfied simultaneously, but they could be at great expense.

For example, "The vehicle shall (1) be able to tow a 2000-pound cargo trailer at highway speed (65 mph); (2) accelerate from 0 to 60 mph in less than 9.5 seconds."

It would be possible but expensive to build a car that satisfied both requirements simultaneously. Similarly, most radios can play AM or FM stations, but not both simultaneously.

For some systems and requirements it is necessary to specify states and for others it is not. Botta et al. (2006) explain when observable states are necessary and when they are not.

Are Your Lights On? However, as with everything, you can take identification of states too far, as illustrated by the following, which is probably a true story. We first saw it in Gause and Weinberg (1990).

Recently, the highway department tested a new safety proposal. The department asked motorists to turn on their headlights as they drove through a tunnel. However, shortly after exiting the tunnel the motorists encountered a scenic-view overlook. Many of them pulled off the road to look at the reflections of wildflowers in pristine mountain streams and snow-covered mountain peaks 50 miles away. When the motorists returned to their cars, they found that their car batteries were dead, because they had left their headlights on. So the highway department decided to erect signs to get the drivers to turn off their headlights.

First they tried “Turn your lights off.” But someone said that not everyone would heed the original request to turn their headlights on and, for these drivers, it would be impossible to turn their headlights off.

So they tried “If your headlights are on, then turn them off.” But someone objected saying that would be inappropriate if it were nighttime.

So they tried “If it is daytime and your headlights are on, then turn them off.” But someone objected saying that would be inappropriate if it were overcast and visibility was greatly reduced.

So they tried “If your headlights are on and they are not required for visibility, then turn them off.” But someone objected saying that many new cars are built so that their headlights are on whenever the motor is running.

So they tried “If your headlights are on, and they are not required for visibility, and you can turn them off, then turn them off.” But someone objected. . . .

So they decided to stop trying to identify applicable states. They would just alert the drivers and let them make the appropriate actions. Their final sign said, “Are your lights on?”

18. *States Assumptions.* All assumptions should be stated. Unstated bad assumptions are one cause of bad requirements.

19. *Usage of Shall, Should, and Will.* A mandatory requirement should be expressed using the word *shall* (e.g., “The system shall conform to all state laws”). A trade-off requirement can be expressed using *should* (e.g., “The total cost for the car’s accessories should be about 10% of the total cost of the car”). The term *will* is used to express a declaration of intent on the part of a contracting agency, to express simple future tense, and for statement of fact (e.g., “The resistors will be supplied by an outside manufacturer”) (Grady, 1993).

20. *Avoids Certain Words.* The words *optimize, maximize, and minimize* should not be used in stating requirements, because we could never prove that we had achieved them. Do not use the word optimize, because you will not be able to prove that a resulting system is optimal. For the same reason, use the words maximize and minimize gingerly. They should not be used in mandatory requirements, but they might appear in trade-off requirements. Never use the words *always* and *never*. Do not use the word *simultaneous* because it means different things to different people. To a physicist, it might mean within a femtosecond, to a computer engineer, on the same clock cycle, to a paleontologist studying the extinction of the dinosaurs, within the same millennium. Use of some words are deprecated. To some people, *operational* means high-level description of behavior. To others, it means low-level keystroke-like activities. Generally, you should avoid adverbs.

21. *Might Vary in Level of Detail.* The amount of detail in the requirements depends on the intended supplier. For in-house work or work to be done by a supplier with well-established systems engineering procedures, the requirements can be written at a high level. However, for outside contractors with unknown systems engineering capabilities, the requirements might be broken down to a very fine level of detail.

Requirements also become more detailed with time. In the beginning, the requirements should describe a generic process that many alternative designs could satisfy. As time progresses, the problem will become better understood, the acceptable number of alternatives will decrease, and the requirements should become more detailed.

22. *Contains Date of Approval.* The name of the approver, the date of approval and the date of the last change should be included in each requirement.

23. *States Its Rationale.* Although it is seldom done, it would be nice if each requirement stated why it was written and what it was supposed to ensure.

24. *Respects the Media.* Newspaper journalists quote out of context, and headlines do not reflect the content of their stories. It is important to write each requirement so that it cannot spark undue public criticism of your customer or their project.

25. *Distinguishes Number.* Be careful with modal words such as shall, should, and will, because they remove number distinction. For example, "Moving vehicles shall be prohibited on these premises". This could mean "Moving of vehicles is prohibited" or "Vehicles that move are prohibited."

26. *Consistent.* The set of requirements should not contain contradictory statements or conditions.

27. *May Use Parameters.* Using parameters rather than fixed numbers will make requirements more reusable. For example, we could write "The radio shall amplify the signal to produce POWER-OUTPUT between FREQUENCY-RANGE," where phrases in capital letters are parameters that could be instantiated to numbers such as 115 watts and 50 and 5000 Hz.

4.5 CHARACTERIZATION OF REQUIREMENTS

There are many independent, orthogonal characterizations of system requirements. Four of these are types, sources, expressions, and input–output relationships. A summary of these characterizations follows.

4.5.1 Types of Requirements

There are two types of system requirements: mandatory and trade-off, as illustrated in Figure 4.2. The *trade-off requirement* is: "The bridge deck should be at the same level as the road surface, within 95% is acceptable". And the *mandatory requirement* is: "The bridge deck shall stretch from bank to bank (95% is not acceptable)."

Mandatory Requirements

1. Specify the necessary and sufficient capabilities that a system must have in order to be acceptable;
2. Use the words *shall* and *must*, although the use of *must* is now deprecated;
3. Are passed or failed with no in between (do not use scoring functions); and
4. Should not be included in trade-off studies.

The following is a typical mandatory requirement: "The system shall not violate federal, state, or local laws". After identifying the mandatory requirements, systems engineers propose alternative candidate designs, all of which satisfy the mandatory requirements. Trade-off requirements are then evaluated to determine the *best* designs.

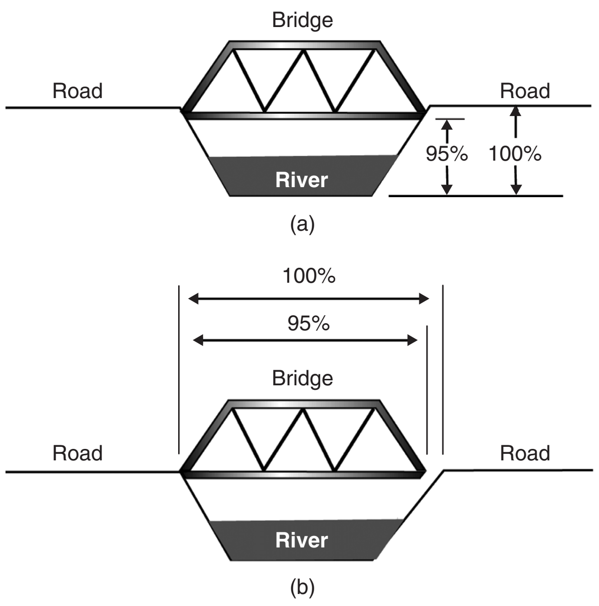

Figure 4.2 Trade-off (a) and mandatory (b) requirements. (Copyright © 1995, A. T. Bahill, from http://www/sie.arizona.edu/sysengr/slides/. Used with permission.)

Tradeoff Requirements

1. State conditions that would make the customer happier and are expressed with the words *shall* or perhaps *should* (often a significant reward or incentive is attached to how well a performance or trade-off requirement is satisfied);
2. Should be described by scoring (utility) functions (Daniels et al., 2001) (see Fig. 4.4) or measures of effectiveness;
3. Should be evaluated with multicriterion decision-making techniques, because none of the feasible alternatives is likely to optimize all the criteria; and
4. There will be trade-offs among these requirements (Smith et al., 2007).

The following is a typical trade-off requirement: “Dinner should have items from each of the five food groups: grains, vegetables, fruits, milk, and meat.”

Figure 4.3 shows an example of such a trade-off in the investigation of alternative laser printers. Many printers were below and to the left of the circular arc. They were clearly inferior and were dismissed. Three printers lay on the circular arc: they were the best. No printers were in the infeasible region above and to the right of the circular arc. The question now becomes: “Which of these three printers is the best?” With the present data, there is no answer to that question. The customer will have to say which trade-off requirement (or evaluation criterion) is more important before an answer can be obtained. Moving from one alternative to another will improve at least one criterion and worsen at least one criterion; that is, there will be trade-offs. An arc like this (or a surface when there are more than two criteria) is called a Pareto optimal contour.

4.5.1.1 Relationships Between Mandatory and Trade-off Requirements Sometimes there is a relationship between mandatory and trade-off requirements: a mandatory

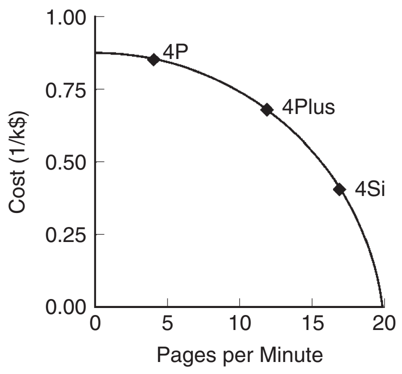

Figure 4.3 A typical trade-off between trade-off requirements.

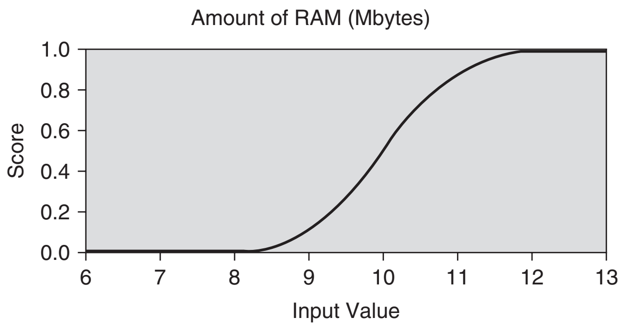

Figure 4.4 A scoring function for the amount of random access memory (RAM).

requirement might be an upper or lower threshold of a trade-off requirement. For example, for one computer program 8 Mbytes of random access memory (RAM) is required, but 12 Mbytes is preferred. But sometimes it will be better to let the mandatory requirement be the baseline value for a trade-off requirement.

4.5.1.2 Scoring Functions Scoring (utility) functions reflect how well each requirement has been met (Daniels et al., 2001). An input value is put into the scoring function and a normalized output score is returned. Higher scores make the customer happier. The use of scoring functions allows different criteria to be compared and traded off against each other (See Fig. 4.4). In other words, scoring functions allow apples to be compared to oranges and nanoseconds to be compared to billions of dollars. A simple program, written by Tom Rogers, that creates graphs such as these is available free at http://www.sie.arizona.edu/sysengr/slides/SSF.zip. It is called the Wymorian Scoring Function tool.

Kano (1993) described three types of requirements: dissatisfiers (mandatory), satisfiers (trade-off), and delighters, which go beyond what the customer expects.

4.5.2 There Are Many Sources of Requirements

We have just analyzed requirements by dividing them into two types: trade-off and mandatory. We will now dissect requirements according to their source. That is, where do requirements come from? We might have called this section “Categories of Requirements” or a “Requirements Taxonomy.”

In this section, we list more than two dozen sources of requirements. However, Wymore (1993) says that only the first six sources are necessary: input–output, technology, performance, cost, trade-off, and system test. He says all of the other sources can be put into one of these six. Grady (1993) says we should have only five sources: functional, performance, constraints, verification, and programmatic. He thinks that most of our sources are constraints. The EIA-632 Standard on Systems Engineering says there are only three: functional, performance, and constraints. Project managers say that there are only three: cost, schedule, and performance (Kerzner, 1995). The Unified Modeling Language (UML) says that there are three categories of requirements: functional requirements, nonfunctional performance requirements, and supplemental requirements that are spread throughout the system. We leave it to the reader to decide whether our list of sources can be condensed.

4.5.2.1 Functions Functional requirements describe the functions that the system must provide (e.g., “Amplify the input signal”).

4.5.2.2 Input–Output Most functional requirements describe an input–output relationship. The amplify function could be stated as “The ratio of the output to the input at 10 kHz shall be +20 dB.” Wymore (1993) maintains that functional requirements are a subset of input–output requirements. A well-stated input–output requirement describes a function. The above input–output requirement describes the function “Amplify the input signal.” The functional requirement, “The system shall fasten pieces of paper,” is covered by the input–output requirement, “The system shall accept 2 to 20 pieces of $8\frac{1}{2}$ by 11 inch, 20-pound paper and secure them so that the papers cannot get out of order.” One function of an automobile is to accelerate. The input is torque (perhaps developed with an engine) and the output is a change in velocity.

4.5.2.3 Technology The technology requirement specifies the set of components—hardware, software, and bioware—that is available to build the system. The technology requirement is usually defined in terms of types of components that *cannot be used*, that *must be used*, or both. For example, Admiral Rickover required that submarine nuclear reactor instrumentation be done solely with magnetic amplifiers. Your purchasing department will often be a source of technology constraints.

4.5.2.4 Performance Performance requirements include quantity (how many, how much), quality (how well), coverage (how much area, how far), timeliness (how responsive, how frequent), and readiness (reliability, availability). Functional requirements often have associated performance requirements: for example, “The car shall accelerate from 0 to 60 mph in 6.5 seconds or less.” Performance is an attribute of products and processes. Performance requirements are initially defined through requirements analyses and trade-off studies using customer need, objective, and/or requirements statements.

4.5.2.5 Cost There are many types of cost, such as labor, resources, and monetary cost. Examples of cost requirement are: “The maximum purchase price shall be \$10,000” and “The maximum total life cycle cost shall be \$18,000.”

4.5.2.6 Trade-off Trade-offs between performance and cost are defined as the different relative value assigned to each factor. A simple trade-off might add the performance and cost criteria with associated weights; for example, the performance criterion may have a weight of 0.6, and the cost criterion may be given a weight of 0.4 (Wymore, 1993; Chapman et al., 1992). However, the trade-off should not be limited to performance and cost: there are many other criteria that should be included in the trade-off.

The trade-off requirement is a description of how the data in a trade-off study are going to be combined (Daniels et al., 2001). The summation of weighted scores (Botta and Bahill, 2007) is one example.

$$\begin{aligned} \mathit{Alternative\ rating} = {} & \mathit{Weight}_{\text{preformance}} \times \mathit{Score}_{\text{performance}} + \mathit{Weight}_{\text{cost}} \times \mathit{Score}_{\text{cost}} \\ & + \mathit{Weight}_{\text{risk}} \times \mathit{Score}_{\text{risk}} \end{aligned}$$

Another example is the benefit−cost ratio:

$$\mathit{Benefit{-}cost\ ratio} = \mathit{Wt}_{\text{Ratio}} \frac{\mathit{Benefit}^{\mathit{Wt}_{\text{Benefit}}}}{\mathit{Cost}^{\mathit{Wt}_{\text{Cost}}}}$$

4.5.2.7 System Test Early in the design process, it should be stated how the final system will be tested. The purpose of the system test is to verify that the design and the system satisfy the requirements. For example, in an electronic amplifier, a 3-mV, 10-kHz sinusoid will be applied to the input, and the ratio of output to input will be calculated.

Currently, we find systems that can be tested and diagnosed over the Internet or over phone lines. A copy machine that does faxes is hooked up to the phone line. Therefore, the supplier could easily query or diagnose the machine on a regular basis over the phone system. The supplier could monitor use and provide just-in-time service for toner and paper. Implementing such testing involves a trade-off of cost with customer convenience and service. This type of testing is great for equipment at remote or unmanned sites.

4.5.2.8 Built-in Self-Test New designs should require that the system test itself. It should perform built-in self-tests (BiST) upon request and whenever it is not servicing one of its actors. Personal computers do built-in self-tests every time they are turned on: the problem is that most people never turn them off.

4.5.2.9 Company Policy Company policy might create requirements. Learjet Inc. has stated, “We will make the airframe, but we will buy the jet engines and the electronic control systems”. Raytheon should require that all missiles have the capability of being disabled by the Pentagon. (This would prevent Stinger missiles from shooting down our airplanes.) Or, “Our company will not deliver anything to the customer that we are unable to test.”

4.5.2.10 Business Practices Corporate business policies might require work breakdown structures, PERT charts, quality manuals, environmental safety and health plans, or a certain return on investment.

4.5.2.11 Systems Engineering Systems engineering might require that every transportable memory device (e.g., floppy disk, Zip, Jaz, Bernoulli, CD ROM, DVD or memory stick) have a Readme file that describes the author, date, contents, software program, and version (e.g., Visio 2007 or Excel 2007).

4.5.2.12 Project Management Performance, cost, and schedule are natural requirements to come from project managers. This might include rewards for early finish and penalties for schedule overruns.

Access to source code for all software might be a project management requirement. It takes time and money to install new software. This investment would be squandered if the supplier went bankrupt and the customer could no longer update and maintain the system. Therefore, most customers would like to have the source code. However, few software houses are willing to provide source code, because it might decrease their profits and complicate customer support. When there is any possibility that the supplier might stop supporting a product, the source code should be provided and placed in escrow. This source code remains untouched as long as the supplier supports the product. But if the supplier ceases to support the product, the customer can get the source code and maintain the product in-house. Therefore, placing the source code in escrow can be a requirement.

4.5.2.13 Marketing The marketing department wants features that will delight the customer. Kano (1993) calls them exciters. They are features that customers did not know they wanted. In the 1970s, IBM queried customers to discover their personal computer (PC) needs. No one mentioned portability, so IBM did not make it a requirement. Compaq made a portable PC and then a laptop, dominating those segments of the market. In the 1950s, IBM could have bought the patents for Xerox's photocopy machine. But they did a market research study and concluded that no one would pay thousands of dollars for a machine that would replace carbon paper. They did not realize that they could delight their customers with a machine that provided dozens of copies in just minutes. The United States Air Force did not know that they wanted a stealth airplane until after the engineers explained that they could do it for the F-117.

4.5.2.14 Manufacturing Processes Sometimes we might require a certain manufacturing process or environment. We might require our semiconductor manufacturer to have a Class 10 clean room. Someone might specify that quality function deployment (QFD) be used to help elicit customer desires (although this would be in bad form, because it states how not what). Recently, minimization of the waste stream has become a common requirement. Annealing swords for a certain time and at a certain temperature could be a requirement.

4.5.2.15 Design Engineers Design engineers impose requirements on the system. These are the "build to," "code to," and "buy to" requirements for products and "how to execute" requirements for processes. These are often called derived requirements, because the design engineers derive them from the customer requirements.

4.5.2.16 Reliability Reliability could be a performance requirement, or it could be broken out separately. It could be given as a mean time between failures or as an operability probability. But, it should be testable: for example, do not say "The system shall be available 99.9% of the time," rather say "The system shall be available 99.9% of the time in a one week test period."

4.5.2.17 Safety Some requirements may come from safety considerations. These may state how the item should behave under both normal and abnormal conditions.

4.5.2.18 The Environment Concern for the environment will produce requirements, such as forbidding the use of chlorofluorocarbons (CFCs) or trichloroethylene (TCE).

4.5.2.19 Ethics Ethics requires physicians to obtain informed consent before experimenting on human subjects.

4.5.2.20 Intangibles Sometimes the desires of the customer will be hard to quantify for intangible items such as aesthetics, national or company prestige (e.g., putting a man on the moon in the Apollo project), ulterior motives such as trying to get a foot in the door using a new technology (e.g., the stealth airplanes), or starting business in a new country (e.g., China).

4.5.2.21 Common Sense Many requirements will not be stated because they are believed to be common sense. For example, characteristics of the end user are seldom stated. If we were designing a computer terminal, it would not be stated that the end user would be a human with two hands and ten fingers. Common sense also dictates that "Computers shall pass diagnostic tests after being stored for 24 hours at 70 °C (163 °F)." Furthermore, we do not write that there can be no exposed high voltage conductors on a personal computer, but it certainly is a requirement. Many of these requirements can be found in de facto standards.

4.5.2.22 Laws or Standards Requirements could specify compliance with certain laws or standards, such as the National Electrical Code, local building codes, EIA-632, ISO-9000, IEEE 1220, ISO-15288, or level 3 CMMI.

4.5.2.23 The Customer Some requirements are said to have come from the customer, such as statements that define the expectations of the system in terms of mission or objectives, environment, constraints, and measures of effectiveness. These requirements are defined from a validated needs statement (Customer's Mission Statement), from acquisition and program decision documentation, and from mission analyses.

4.5.2.24 Legacy Requirements Sometimes the existence of previous systems creates requirements. For example, "Your last system was robust enough to survive a long trip on a dirt road, so we expect your new system to do the same." Many new computer programs must be compatible with COBOL, because so many COBOL business programs are still running. Could you read data archived on a $5\frac{1}{4}$ inch floppy disk? Legacy requirements are often unstated.

4.5.2.25 Existing Data Collection Activities If an existing system is similar to the proposed new system, then existing data collection activities can be used to help discover system requirements, because each piece of data that is collected should be traceable to a specific system requirement, as shown in Figure 4.1. Often it is difficult to make a measurement to verify a requirement. It might be impossible to meet the stated accuracy. Trying to make a measurement to verify a requirement might reveal more system requirements.

4.5.2.26 Human Abuse Systems should be required to withstand human abuse, such as standing on a computer case or spilling coffee on the keyboard. Systems should prevent human error; for example, "Are you sure you want to delete that file?" Systems should ameliorate human error if it does occur (e.g., online real-time spell checking).

4.5.2.27 Political Correctness The need to be politically correct mandates many (often expensive) requirements.

4.5.2.28 Material Acceptance You must write requirements describing how raw materials, parts, and commercial off-the-shelf items will be inspected when they are received.

4.5.2.29 Other Sources There are many other sources of system requirements, such as human factors, the environment (e.g., temperature, humidity, shock, vibration, etc.), the end user, the operator, potential victims, management, company vision, future expansion, schedule, logistics, politics, the U.S. Congress, public opinion, business partners, past failures, competitive intelligence, liability, religion, culture, government agencies (e.g., DoE, DoD, OSHA, FAA, EPA), availability, maintainability, compatibility, service, maintenance, field support, warrantees, need to provide training, competitive strategic advantage, time to market, time to fill orders, inventory turns, accident reports, deliverability, reusability, future expansion, politics, society, standards compliance, standards certification (e.g., ISO 9000), effects of aging, the year 2000 problem, user friendly, weather (e.g., must be installed in the summer), security as in government classification, security as in data transmission, it must fit into a certain space, secondary customer, retirement, and disposal.

4.5.3 There Are Many Ways to Express Requirements

For some purposes, the best expression of the requirements will be a narrative in which words are organized into sentences and paragraphs. Such documents are often called operational concept descriptions or use case models. But all descriptions in English will have ambiguities, because of both the language itself and the context in which the reader interprets the words. Therefore, for some purposes, the best description of a system will be a list or string of *shall* and *should* statements. Such a list would be useful for acquisition or acceptance testing. However, it is still very difficult to write with perfect clarity so that all readers have the same understanding of what is written. Other modalities that can be used instead of written descriptions include:

- A model
- A prototype
- A user's manual

- Input–output trajectories
- Sequence diagrams
- Use cases
- Computer requirements databases

The big advantage of nonverbal expressions is that they can be rigorous and executable. This helps point out contradictions and omissions. Natural language expressions can be ambiguous. Does “turn up the air conditioner,” mean turn up the thermostat (making the room warmer), or turn up the air conditioning power (making the room colder)? Nonverbal expressions also allow you to perform a sensitivity analysis of the set of requirements to learn which requirements are the real cost drivers (Karnavas et al., 1993; Smith et al., 2007).

A systems engineering design process that is gaining popularity is model-based systems engineering (Wymore, 1993). In this process, the systems engineer first develops a model of the desired system. The model can then be run to demonstrate the desired system behavior. Thus, the model also captures the requirements.

4.5.3.1 A Prototype Expresses Requirements A publicly assessable prototype can express the system requirements, as they are currently understood. This technique is very popular in the software community, where a computer can be placed in the building lobby. Of course, many functions of the final system will not be implemented in the prototype; instead, there will be a statement of what the functions are intended to do. A publicly assessable prototype is easy to update, and it helps everyone understand what the requirements are.

The purpose of building a prototype is to reduce project risk. Therefore, the first functions that are prototyped should be (but usually are not) the most risky functions (Eb Rechtin, personal communication; Botta and Bahill, 2007).

4.5.3.2 Consider Bizarre Alternatives During concept exploration, encourage consideration of bizarre alternatives. Studying unusual alternatives leads to a better and deeper understanding of the requirements by both the systems engineer and the design engineer. Explore unintended paths through the system workflow. Document unintended functions performed by the system. Investigate unintended uses of the system. Studying models and computer simulations will also help you understand the requirements. Concept exploration is one of the most fruitful phases in requirements discovery.

4.5.3.3 Preparing the Users Manual Flushes Out Requirements The users manual should be written by future users early in the system design process (Shand, 1994). This helps get the system requirements stated correctly and increases user “buy in.”

4.5.4 Input and Output Trajectories

Arguably, the best way to describe the desired behavior of a system and thereby discover system requirements is to create typical sequences of events (or scenarios) that the proposed system will go through. Typically, a sequence of input values as a function of time (called an input trajectory) is described and an acceptable system behavior is given. Sometimes this system behavior is given as a sequence of output values (called an output trajectory). Dozens, or perhaps hundreds (but hopefully not thousands), of these trajectories will be needed for a complete description of the system. Such descriptions of input and output behavior as a function of time have been called behavioral scenarios, strings, trajectories, threads, operational scenarios, logistics, interaction diagrams, and sequence diagrams. The alternate flows help you recognize and deal with unintended as well as intended inputs.

4.5.4.1 Sequence Diagrams A sequence diagram for an automated teller machine (ATM) is shown in Figure 4.5. The basis of these diagrams is to list the system's objects (called Lifelines) along the top of the diagram. Then, with time running from top to bottom, list the messages or commands that are exchanged between the objects (Object Management Group, 2007). Alternatively, the arrows can be labeled with data

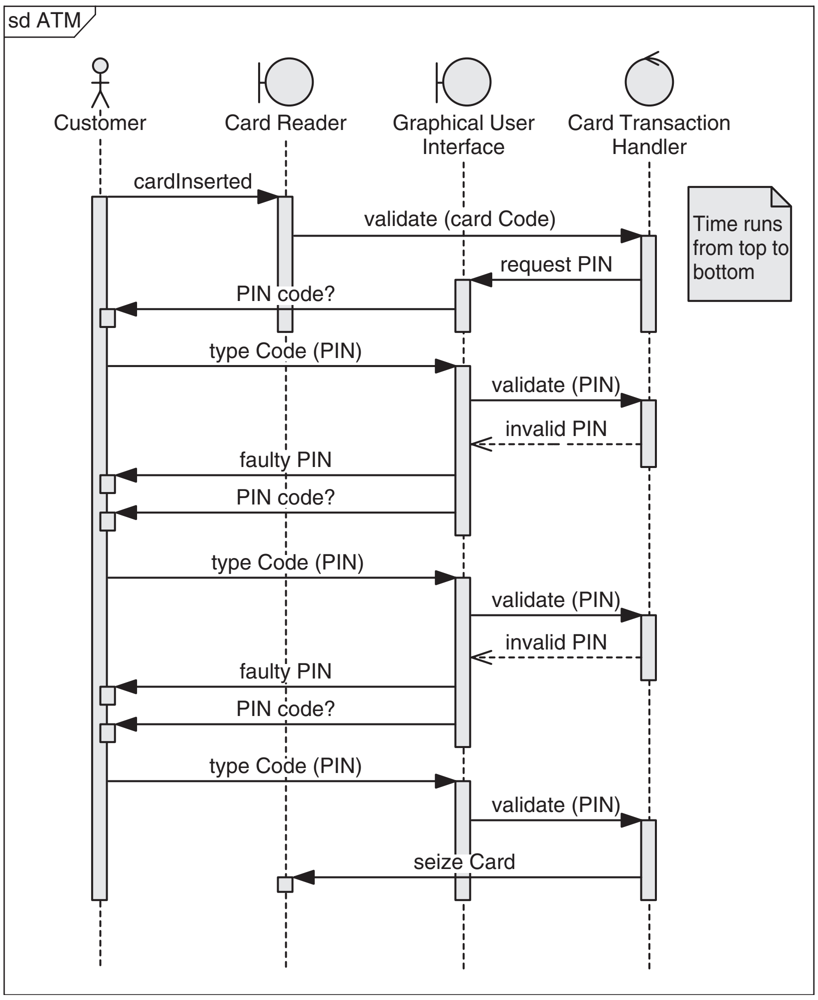

Figure 4.5 Sequence diagram for an incorrect personal identification number (PIN) sequence of events. (Copyright © 2007, A. T. Bahill, from http://www/sie.arizona.edu/sysengr/slides/. Used with permission.)

that are exchanged between the components or the functions that are performed. These ATM examples were derived using object-oriented modeling. This technique relies on collecting a large number of sequence diagrams. This collection then describes the desired system behavior. Additional scenarios can be incrementally added to the collection. Sequence diagrams are easy for people to describe and discuss, and it is easy to transform them into a system design.

Wrong PIN Input Sequence

1. The Customer inserts a bankcard; the Card Reader sends the card's information to the Card Transaction Handler, which detects that the card is valid (if no message is returned, the card is assumed valid).
2. The Card Transaction Handler instructs the Graphical User Interface (GUI) to display a message requesting the customer's Personal Identification Number (PIN).
3. The GUI requests the PIN and the customer types in his/her PIN, which is then passed to the Card Transaction Handler.
4. The Card Transaction Handler checks if the PIN is correct. In this scenario, it is not, and the GUI is instructed to inform the customer that the PIN is invalid.
5. The customer is then asked to input his/her PIN again and step 4 is repeated.
6. If the customer has not supplied the correct PIN in three attempts (as is the case in this scenario), the Card Reader is instructed to keep the card and the session is terminated.

Sequence diagrams were first published by Richard Feynman (1949). Feynman diagrams are a visual way of expressing what is happening. He created them because they are easier to understand than their accompanying equations.

4.5.4.2 Input–Output Relationships Wymore (1993) shows the following six techniques for writing input–output relationships. These techniques have different degrees of precision, comprehensibility, and compactness.

1. For each input value, produce an output value. For example, multiply the input by 3:

   ```
   output(t+1) = 3 * input(t)
   ```

2. For each input string, produce an output value. For example, compute the average of the last three inputs:

   ```
   output(t+1) = (input(t−2) + input(t−1) + input(t))/3
   ```

3. For each input string, produce an output string. For example, collect inputs and label them with their time of arrival:

   ```
   For an input string of 1, 1, 2, 3, 5, 8, 13, 21, the
   output string shall be (1,1), (2,1), (3,2), (4,3), (5,5),
   (6,8), (7,13), (8,21). All strings are finite in length.
   ```

4. For each input trajectory, produce an output trajectory. For example, collect inputs and label them with their time of arrival:

```
For an input trajectory of 1, 1, 2, 3, 5, 8, 13, 21, ...
the output trajectory would be (1,1), (2,1), (3,2), (4,3),
(5,5), (6,8), (7,13), (8,21) ... A trajectory may be
infinite in length.
```

5. For each state and input, produce a next state and next output. For example, design a Boolean system where the output is asserted whenever the input bit stream has an odd number of ones. This Odd Parity Detector can be described as:

```
Z1 = (SZ1, IZ1, OZ1, NZ1, RZ1), where
SZ1 = {Even, Odd},/* The 2 states are named Even and Odd. */
IZ1 = {0, 1},/* A 0 or a 1 can be received on this input
port. */
OZ1 = {0, 1},/* The output will be 0 or 1. */
NZ1 = {((Even, 0), Even),/* If the present state is Even and
the input is 0, then the next state will be Even. */
((Even, 1), Odd), ((Odd, 0), Odd), ((Odd, 1), Even)},
RZ1 = {(Even, 0), (Odd, 1)}/* If the state is Even the
output is 0, if the state is Odd the output is 1. */
```

6. Most of this chapter has focused on using qualitative descriptions, which includes words, sentences, paragraphs, blueprints, pictures, and schematics.

4.6 THE REQUIREMENTS DEVELOPMENT AND MANAGEMENT PROCESS

The requirements process shown in Figure 4.6 should be interpreted figuratively not literally: it only contains major flows. It shows 16 tasks, so there should be 16! feedback paths. For example, use cases cannot be written without information about the stakeholders, customer needs, and a problem statement; verification and validation cannot be done without knowledge of the requirements.

The first steps in discovering system requirements are identifying the stakeholders, understanding the customers' needs, stating the problem, and identifying the needed capabilities of the system.

4.6.1 Identify Stakeholders

The first step in developing requirements is to identify the stakeholders. The term *stakeholder* includes anyone who has a right to impose requirements on the system. This includes end users, operators, maintainers, bill payers, owners, regulatory agencies, victims, and sponsors. All facets of the stakeholders must be kept in mind during system design. For example, in evaluating the cost of a system, the total life-cycle cost and the cost to society should be considered. Frequently, the end user does not fund the cost of development. This often leads to products that are expensive to own, operate, and maintain over the entire life of the product, because the organization funding development saves a few dollars in the development process. Therefore, it is imperative that the systems engineer understands this conflict and exposes it. The sponsor and user can then help trade off the development costs against the cost to

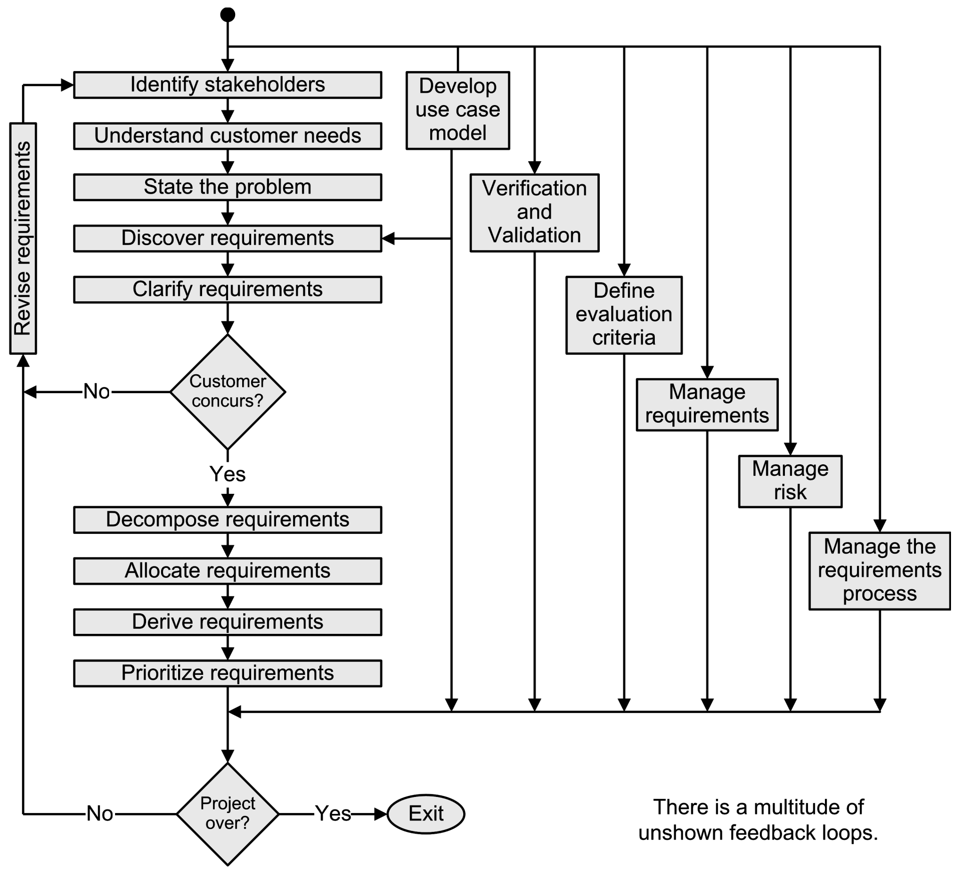

Figure 4.6 The requirements process. (Copyright © 2006, A. T. Bahill, from http://www/sie.arizona.edu/sysengr/slides/. Used with permission.)

use and maintain. Total life-cycle costs are significantly larger than initial costs. For example, in one of their advertisements, Compaq proclaimed that “80% of the lifetime cost of your company’s desktops comes after you purchase them.” In terms of the personal computer, if total life-cycle costs were \$5000, purchase cost would have been \$1000 and maintenance and operation \$4000.

For large or complex systems, it is important that the systems engineer understands the role of each of the major stakeholders and their motivations, interests, constraints, and outside pressures.

4.6.2 Understand Customer Needs

The system design must begin with a complete understanding of the customers’ needs. If the sponsor, owner, and user are different, then the systems engineer must know and understand the needs of each. For a rental car agency, the sponsor could be corporate management, the owner of the cars could be the local airport office, and the user could be the customer renting the car. Each of these three has needs; some are common and some are unique. The information necessary to begin a design usually comes from the mission statement, the concept of operation, business model, preliminary studies, and specific customer requests. Usually the customer is not aware of what is needed. Systems engineers must enter the customer's environment, discover the details, and explain them. Flexible designs and rapid prototyping facilitate identification of details that might have been overlooked. Talking to the customer's customer and the supplier's supplier can also be useful. This activity is frequently referred to as mission analysis.

It is the systems engineer's responsibility to ensure that all information concerning the customer's needs is collected. The systems engineer must also ensure that the definitions and terms used have the same meaning for everyone involved. Several direct interviews with the customer are necessary to ensure that all of the customer's needs are stated and that they are clear and understandable. The customer might not understand the needs; he/she may be responding to someone else's requirements. Often, a customer will misstate his/her needs; for example, a person might walk into a hardware store and say he needs a half-inch drill bit. But what he actually needs is a half-inch *hole* in a metal plate, and a chassis-punch might be more suitable.

During the first half of the 20th century, the Indian and Harley Davidson motorcycle manufacturers were fierce competitors. Then during World War II, the U.S. Army asked for a 500-cc motorcycle. Indian made one for them. Harley Davidson thought that the Army would not be satisfied with a 500-cc motorcycle, so they built a 750-cc motorcycle. Since then, have you heard of an Indian motorcycle?

4.6.3 State the Problem

What is the problem we are trying to solve? Answering this question is one of the systems engineer's most important and often overlooked tasks. An elegant solution to the wrong problem is less than worthless.

Early in the process, the customer frequently fails to recognize the scope or magnitude of the problem that is to be solved. The problem should not be described in terms of a perceived solution. It is imperative that the systems engineer help the customer develop a problem statement that is completely independent of solutions and specific technologies. Solutions and technologies are, of course, important; however, there is a proper place for them later in the systems engineering process. It is the systems engineer's responsibility to work with the customer, asking the questions necessary to develop a complete picture of the problem and its scope. The U.S. Air Force did not know that they wanted a stealth airplane until after the engineers showed that they could do it. During concept exploration, encourage consideration of bizarre alternatives. This will help you understand the requirements better. Likewise, studying models and computer simulations will help you understand the requirements. Understanding the requirements is one of the most fruitful phases in requirements discovery.

Defining what the system is supposed to do could be done in terms of a hierarchy of functions. But recently it is being stated in terms of a hierarchy of capabilities the system must have.

4.6.4 Develop Use Case Models

Modern system design is usually use case based. The use cases capture the required functional behavior of the system. A use case is an abstraction of a required behavior of a system. A use case produces an observable result of value to the user. A typical system will have many use cases, each of which satisfies a goal of a particular user. Each use case describes a sequence of interactions between one or more users and the system. A use case narrative is written in a natural language. It is written as a sequence of steps with a main success scenario and alternate flows. This sequence of interactions is contained in a use case report and the relationships between the system and its users are portrayed in use case diagrams. Development of requirements from the use case models is explained in detail in the last section of this chapter.

4.6.5 Discover Requirements

The systems engineer must consult with the customer to discover the system requirements. The systems engineer must involve the customer in the process of defining, clarifying, and prioritizing the requirements. It is prudent to involve users, bill payers, regulators, manufacturers, maintainers, and other key stakeholders in the process. Requirements are discovered or elicited: they are not given to you. It takes work to find out what the requirements are.

Next, systems engineering must discover the functions that the system must perform in order to satisfy its purpose. The system functions form the basis for dividing the system into subsystems. QFD is useful for identifying system functions (Bahill and Chapman, 1993; Bicknell and Bicknell, 1994).

Although this makes it sound as if requirements are transformed into functions with a waterfall process, that is not the case. The requirements process is highly iterative and many tasks can and should be done in parallel. First, we look at system requirements, then at system functions. Then we reexamine the requirements and then reexamine the functions. Then we reassess the requirements and again the functions, and so on. Identifying the system's functions helps us to discover the system's behavioral requirements.

For each requirement that is discovered, ask yourself why the requirement is needed. This will help you to write the rationale for the requirement, it will help you to prioritize the requirements, and it might help you to negotiate with the customer to eliminate some requirements.

4.6.6 Clarify Requirements

The systems engineer must consult with the customer to ensure that the requirements are correct and complete and to identify the trade-off requirements. As with all systems engineering processes, this process is iterative. The customer should be satisfied that if the requirements are met, then the system will do what it needs to do. This should be done in formal reviews with the results documented and distributed to appropriate parties. These reviews ensure that all the requirements have been met, ensure that the system satisfies customer needs, assess the maturity of the development effort, allow recommending start of the next phase, and facilitate approval to committing additional resources. The systems engineer is responsible for initiating and conducting these reviews.

The system requirements must be reviewed with the customer many times. At a minimum, requirements should be reviewed at the end of the modeling phase, after testing the prototypes, before commencement of production, and after testing production units.

It is the job of the systems engineer to push back on the customer. The systems engineer should try to get requirements removed or relaxed. Perform a sensitivity analysis on the requirements set to determine the cost drivers. Then show the customer how cost can be reduced and performance enhanced if certain requirements are changed. The requirements process should be a continual interactive dialogue with the stakeholders.

The main objectives of these reviews are to find missing requirements, eliminate unneeded requirements, ensure that the requirements have been met, and verify that the system satisfies customer needs. At these reviews, trade-offs will usually have to be made among performance, schedule, and cost. Additional objectives include recommending whether to proceed to the next phase of the project and committing additional resources. The results and conclusions of the reviews should be documented. Again, the systems engineer is responsible for initiating and conducting these reviews.

The following definitions of the most common reviews, based on Sage (1992) and Shishko and Chamberlain (1995), might be useful. No company uses these exact names and reviews. But most companies use similar names and reviews. They are arranged in chronological order. Although these definitions are written with a singular noun, they are often implemented with a collection of reviews. Each system, subsystem, sub-subsystem, and so on will be reviewed and the totality of these constitutes the indicated review.

Mission Concept Review (MCR) (a.k.a. the Mission Definition Review and the Alternate System Review). This is the first formal review. It examines the mission objectives, candidate architecture, top-level functional requirements, measures of effectiveness, interfaces, alternative concepts, and anticipated risks.

System Requirements Review (SRR). This shows that the product development team understands the mission and the system requirements. It confirms that the system requirements are sufficient to meet mission objectives. It ensures that the performance and cost evaluation criteria are realistic, and that the verification plan is adequate. It inspects the requirements flow-down plan. It checks that the selected requirements tracking and management tool is appropriate. It assesses the reasonableness of the risk analysis. At the end of SRR, requirements are placed under configuration management. Changing requirements after SRR will adversely affect schedule and cost.

In the beginning of a project, requirements can be kept in a word document. However, eventually it will be more convenient to move them to a database. In the proposal phase and up through the SRR, the requirements might be kept in an Excel spreadsheet. As the number of requirements exceeds 100, most programs will transfer the requirements to a requirements-specific database.

System Functional Review. This examines the proposed system architecture, the proposed system design, and the allocation of functions to the major subsystems. It reviews information sharing, interface definitions and interface limitations such as security blocks. It also ensures that the verification and risk mitigation plans are complete.

Preliminary Design Review (PDR). This demonstrates that the preliminary design meets all the system requirements with acceptable risk. The design model is presented. System development and verification tools are identified, and the work breakdown structure is examined. Full-scale engineering design begins after this review.

Critical Design Review (CDR). This verifies that the design meets the requirements. The CDR examines the system design in full detail, inspects interface control documents, ensures that technical problems and design anomalies have been resolved, checks the technical performance measures, and ensures that the design maturity justifies the decision to commence manufacturing and coding. Few requirements should be changed after this review.

Test Readiness Review (TRR). This is conducted for each hardware and software configuration item prior to formal test. It determines the readiness and completeness of test procedures and their compliance with test plans and descriptions. It inspects test equipment and facilities. It analyses performance predictions.

Production Readiness Review (PRR). For some systems, there is a long phase when prototypes are built and tested. At the end of this phase, and before production begins, there is a production readiness review. The systems engineer is responsible for designing the product and also the process for producing it. This review primarily inspects the requirements for the production system.

Total System Test (TST). At the end of manufacturing and integration, the system is tested to verify that it satisfies its requirements. Technical performance measures are compared to their goals. The results of these tests are presented at the System Acceptance and Operational Readiness Reviews.

Figure 4.7, based on the IBM Rational Unified Process and Bahill and Daniels (2003), shows the timing of some of these reviews within the total system life cycle. The amount of black ink associated with each activity indicates the amount of time, effort, and money that is consumed.

At various points in the system life cycle and for various reasons, the following reviews are often conducted: Startup Review, Software Specification Review, Test Readiness Review, Peer Reviews, Expert Reviews, and Integration Readiness Review.

At these reviews, it is important to ask why each requirement is needed. This can help eliminate unneeded requirements. It can also help reveal the requirements behind the stated requirements. It may be easier to satisfy the requirements behind the requirements, than the stated requirements themselves.

4.6.7 Decompose Requirements

Requirements decomposition breaks down a requirement into two or more requirements whose total content is equivalent to the content of the original one—just expressed more explicitly or in more detail. The resulting requirement replaces the decomposed requirement, which becomes obsolete, or at least it need not be verified. When a written requirement is compound, it must be decomposed. Here is an example of requirement decomposition. The customer requirement, “When an empty coffeepot is placed in the coffee machine, the camera system shall transmit a digital image to the server” can be decomposed into the following functional requirements:

1. The system shall sense when an empty coffeepot is being placed in the coffee machine.

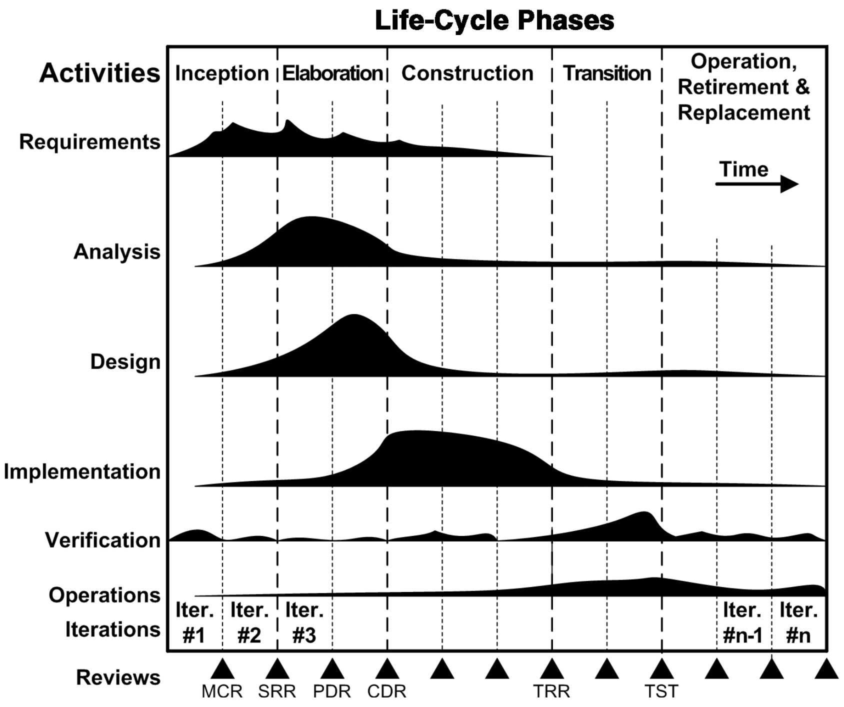

Figure 4.7 Timing of the major reviews. (Copyright © 2006, A. T. Bahill, from http://www/sie.arizona.edu/sysengr/slides/. Used with permission.)

2. The system shall trigger the camera when an empty coffeepot is placed in the coffee machine.
3. The system shall transmit digital images to the server.

Stipulation: Empty coffeepot means one containing less than 6 fluid ounces. Then the original requirement can be deleted.

If the requirements are organized in a Zachman framework (Bahill et al., 2006), then any requirement that spans two or more columns is a candidate for decomposition.

4.6.8 Allocate Requirements

Each functional requirement should be allocated to a particular physical component (Bahill and Botta, 2008). Allocate requirements to hardware, software, bioware, test, and interface components. Allocation is done early in the system life cycle and at a high level in the requirements hierarchy. Allocation affects the architecture. We allocate the system requirements. Allocation is not as important for derived requirements, because their allocation just goes along with their parents' allocation and the architecture is already set before we derive most of these requirements.

4.6.9 Derive Requirements

Derived requirements arise from constraints, consideration of issues implied but not explicitly stated in the customer requirements, factors introduced by the selected architecture, the design, and the developer's unique business considerations. Unlike decomposed requirements, the statements of the derived requirements are different from those of the original requirements. Consequently, the original requirements do not become obsolete.

For example, the U.S. Marine Corp's landing craft must have a range of at least 25 nautical miles so that the mother craft can launch from over the horizon. The maximum duration of a trip from the mother craft to the beach is 1 hour, because trips of over an hour will make most people too seasick to function effectively. Therefore, a derived requirement is "The landing craft shall have a minimum speed of 25 knots."

Design engineers often derive requirements from the customer requirements. Here is a longer example of deriving requirements. A teenage boy might express the *customer needs statement* this way: "Hey, Dad, We need speakers in the car that will make your insides rumble during drum solos." The father would translate this into *the performance requirement:* "For bass frequencies, we need 110 dB of sound output." Then, the systems engineer would convert this into *the functional requirement:* "Amplify the radio's output to produce 115 watts in the frequency range 20 to 500 Hz." Finally, after a trip to the audio shop, the design engineer would transform this into *the design requirement:* "Use Zapco Z100S1VX power amplifiers with JL Audio 12 W1-8 speakers." But this looks like a waterfall process, whereas the requirements process is concurrent and iterative.

4.6.10 Prioritize Requirements

Requirements should be prioritized (Botta and Bahill, 2006, 2007). (1) If the project is budget constrained, prioritization will help you decide which requirements should be implemented and which should be candidates for elimination. (2) If the project is time constrained, prioritization will help you decide which requirements should be implemented first. Often the product is delivered in phases. At each delivery, the system must have testable functionality. Prioritization helps you choose the functions to implement in each phase. (3) Prioritizing scenarios and identifying benefits, costs, and dependencies will help create the system architecture. (4) Prioritization improves customer satisfaction by increasing the likelihood that the customer's most important requirements are implemented and delivered first. Customers like to see their funds being used effectively and wisely. (5) Prioritization will allow you to spend more time and effort reducing risks associated with hard technical problems and key performance parameters. (6) You might want to assign your best people to the highest priority requirements. (7) Prioritizing requirements will help you to manage requirements creep. If requirements being added are high priority, then they may displace some low-priority requirements. (8) Prioritizing requirements will reduce discussion time at meetings and reviews. (9) Prioritizing requirements will help identify the high-priority requirements for which you should create *technical performance measures*, which is a topic in the next section.

4.6.11 Define Evaluation Criteria

Evaluation criteria are used to judge the different designs. Each evaluation criterion must have a fully described unit of measurement. Units of power could be horsepower, for example, and units of cost could be dollars (or inverse dollars if it is desirable to consistently have "more is better" situations). Suppose an evaluation criterion were acceleration; then the unit of measurement could be seconds taken to accelerate from 0 to 60 mph. The units of measurement can be anything, as long as they measure the appropriate criteria, are fully described, and are used consistently for all designs. The value of an evaluation criterion describes how effectively a trade-off requirement has been met. For example, the car went from 0 to 60 in 6.2 seconds. These values are the ones put into the scoring functions, as shown in Figure 4.4, to give the requirements scores, which are in turn used to perform trade-off studies (Smith et al., 2007). Such measurements are made throughout the development of the system.

4.6.11.1 Definitions for Quantitative Measurements Evaluation criteria, measures, and technical performance measures (TPMs) are all used to quantify system performance. These terms are often used interchangeably, but a distinction is useful. Evaluation criteria are used to quantify requirements. Measures are used to help manage a company's processes. TPMs are used to mitigate risk during design and manufacturing.

Performance, schedule, and cost *evaluation criteria* show how well the system satisfies its requirements (e.g., in this test the car accelerated from 0 to 60 in 6.2 seconds). Criteria are often called *measures of performance*, *measures of effectiveness*, or *figures of merit*. Such measurements are made throughout the evolution of the system: based first on estimates by the design engineers, then on models, simulations, and prototypes, and finally on the real system. Criteria are used to quantify system requirements; however, they are also used to help select among alternative designs in trade-off studies, where criteria are traded off: that is, going from one alternative to another increases the value of one criterion while decreasing another.

Measures (which are often confused with *metrics*) are usually related to the process, not the product. Therefore, they do not always relate to specific system requirements. Rather, some measures relate to the company's mission statement and subsequent goals. As an example, a useful measure is the percentage of requirements that change after the System Requirements Review.

Measure. A measure indicates the degree to which an entity possesses and exhibits a quality or an attribute. A measure has a specified method, which when executed produces values (or metrics) for the measure.

Metric. A measured, calculated, or derived value (or number) used by a measure to quantify the degree to which an entity possesses and exhibits a quality or an attribute.

Measurement. A value obtained by measuring, which makes it a type of metric.

Technical performance measures (TPMs) are tools that show how well a system is satisfying its requirements or meeting its goals. TPMs provide assessments of the product and the process through design, modeling, breadboards, prototypes, implementation, and test. For the rest of this section, we only discuss TPMs.

4.6.11.2 Technical Performance Measures Technical performance measures are tools that show how well a system is satisfying its requirements or meeting its goals (Oakes et al., 2006). TPMs provide assessments of the product and the process through design, implementation, and test. They assess the product and its associated process, but they are primarily for the product. TPMs are used to (1) forecast values to be achieved through planned technical effort, (2) measure differences between achieved values and those allocated to the product, (3) determine the impact of these differences, and (4) trigger optional design reviews.

TPMs are critical technical parameters that a project monitors to ensure that the technical objectives of a product will be realized. Typically, TPMs have planned values at defined time increments, against which the actual values are plotted. Monitoring TPMs allows trend detection and correction and helps identify possible performance problems prior to incurring significant cost or schedule overruns.

The purpose of TPMs is to (1) provide visibility of actual versus planned performance, (2) provide early detection or prediction of problems requiring management attention, (3) support assessment of the impact of proposed changes, and (4) help manage risk.

The method for measuring the data for the TPM will vary with life-cycle phase. In the beginning, you will use data from legacy systems, blue-sky guesses, and approximations. Then you can derive data from models and simulations. Later, you will collect data from prototypes. Finally, you will measure data on rudiments of the real system. Even the planned values might not be known at the beginning of the project. The original estimates will be refined by modeling and simulation.

Attributes that could be TPMs for some systems include reliability, maintainability, power required, weight, throughput, human factors, response time, complexity, availability, accuracy, image processing rate, achieved temperature, requirements volatility, speed, setup time, change over time, and calibration time.

4.6.11.3 Characteristics of TPMs TPMs should be created for requirements that satisfy all of the following criteria:

1. High-priority requirements that impact mission accomplishment, customer satisfaction, cost or system usefulness.
2. High-risk requirements that have high probability of not being met. And if they are not met, there is a high probability of project/system failure and great consequences of failure.
3. Requirements where the desired performance is not currently being met. The causes may include the use of new or unproven technology, imposition of additional constraints (e.g., a drastic increase in the number of users), or an increase in the performance target.
4. Requirements where the performance is expected to improve with time, where progress toward a goal is expected.
5. Requirements where the performance can be controlled.
6. Requirements where the program manager is able to trade off cost, schedule, and performance. If TPMs exceed their thresholds and indicate imminent cost overruns or failure to meet performance goals, then the associated requirements should be renegotiated with the customer. Occasionally, TPMs indicate that performance needs are being greatly exceeded, often because of innovative approaches, advances in technology, or materials. This is important to recognize because resources and trade-offs can be redirected to other critical areas or requirements.

TPMs require quantitative data to evaluate how well the system is satisfying its requirements. Gathering such data can be expensive. Because of the expense, not all requirements have TPMs, just the high-priority requirements. As a rule of thumb, less than 1% of requirements should have TPMs: each program might have half a dozen TPMs.

- TPMs should reveal the state of health of the project.
- TPMs should be important.
- TPMs should be relevant.
- TPMs should be relatively easy to measure, analyze, interpret, and communicate with upper management.
- Performance should be expected to improve with time.
- If the measure crosses its threshold, corrective action should be known.
- The measured parameter should be controllable. If the project team cannot change the measured parameter, then do not measure it. For example, in the design of a home air conditioning system, the outside air temperature would be an important design input, but a terrible TPM.
- Management should be able to trade off cost, schedule, and performance.
- TPMs should be documented.
- TPMs should be tailored for the project.

The TPMs can be displayed with graphs, charts, diagrams, figures, or frames, for example, statistical process control charts, run charts, flow charts, histograms, Pareto diagrams, scatter diagrams, check sheets, PERT charts, Gantt charts, line graphs, process capability charts, and pie charts.

As an example, let us consider the design and manufacture of solar ovens. In many societies, particularly in Africa, many women spend as much as 50% of their time acquiring wood for their cooking fires. To ameliorate this sink of human resources, people have been designing and building solar ovens. Let us now examine the solar oven design and manufacturing process that we followed in a freshman engineering design class at the University of Arizona.

First, we defined a TPM for our design and manufacturing process. When a loaf of bread is finished baking, its internal temperature should be 95 °C (203 °F). To reach this internal temperature, commercial bakeries bake the loaf at 230 °C (446 °F). As initial values for our oven temperature TPM, we chose a lower limit of 100 °C, a goal of 230 °C, and an upper limit of 270 °C. The tolerance band shrinks with time as shown in Figure 4.8.

In the beginning of the design and manufacturing process, our day-by-day measurements of this metric increased because of finding better insulators, finding better glazing materials (e.g., glass and Mylar), sealing the cardboard box better, aiming at the sun better, and so on.

At the time labeled "Design Change-1," there was a jump in performance caused by adding a second layer of glazing to the window in the top of the oven. This was followed by another period of gradual improvement as we learned to stabilize the two pieces of glazing material.

At the time labeled "Design Change-2," there was another jump in performance caused by a design change that incorporated reflectors to reflect more sunlight onto the

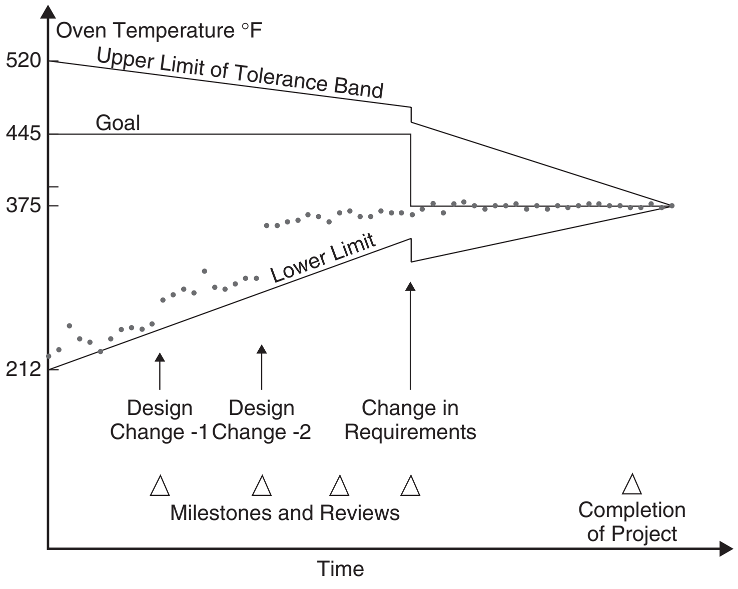

Figure 4.8 Illustration for a technical performance measure.

window in the oven top. This was followed by another period of gradual improvement as we found better shapes and positions for the reflectors.

But, in this case, it seemed that we might not attain our goal. Therefore, we reevaluated the process and the requirements. Bread baking is a complex biochemical process that has been studied extensively: millions of loaves have been baked each day for the last 4000 years. These experiments have revealed the following consequences of insufficient oven temperature:

1. Enzymes are not deactivated soon enough, and excessive gas expansion causes coarse grain and harsh texture.
2. The crust is too thick, because of drying caused by the longer duration of baking.
3. The bread becomes dry, because prolonged baking causes evaporation of moisture and volatile substances.
4. Low temperatures cannot produce carmelization, and crust color lacks an appealing bloom.

After consulting some bakers, our managers decided that 190 °C (374 °F) would be sufficient to avoid the above problems. Therefore, the requirements were changed at the indicated spot and our TPM was then able to meet our goal. Of course, this change in requirements forced a review of *all* other requirements and a change in many other facets of the design. For example, the duration versus weight tables had to be recomputed.

If sugar, eggs, butter, and milk were added to the dough, we could get away with temperatures as low as 175 °C (347 °F). But we decided to design our ovens to match the needs of our customers, rather than try to change our customers to match our ovens.

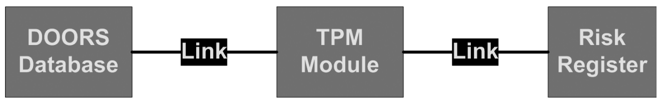

Figure 4.9 Linkages of the TPM module. (Copyright © 2004, A. T. Bahill, from http://www/sie.arizona.edu/sysengr/slides/. Used with permission.)

We have stated earlier that when it appears that a TPM cannot be met, the program manager must readjust performance requirements, cost, and schedule. In this example, we renegotiated the requirements.

If the TPM is being displayed with a graph, then the graph might contain the goal, upper and lower thresholds, planned values, and actual values. In Figure 4.8, the planned values were coincident with the lower limit and were not called out separately.

The requirements management module should have an attribute named TPM. The name of each TPM should be entered in the attribute field of the appropriate requirement and this should be linked to the TPM Module as shown in Figure 4.9. Each TPM should also be put in the project's Risk Register and be evaluated monthly.

Each TPM must be linked to a requirement (or a work breakdown structure element), have quantitative values, have a risk level, and have a priority. You should only create TPMs for requirements where you can take action to cause a change in the quantity or result being measured.

In summary, TPMs are used to identify and track performance requirements that are program critical. TPMs are used to establish the appropriate design emphasis and design criteria and identify levels of technical risk. TPM are collected and tracked against project design objectives in the project's risk register. TPMs should be created for high-priority requirements that impact mission accomplishment, customer satisfaction, system usefulness, and where performance improves with time, where performance can be controlled, and where management can trade off cost, schedule, and performance.

4.6.12 Manage Requirements

The purpose of requirements management is to (1) track and control requirements, requirement changes, and reallocations, (2) establish system requirements traceability to lower level design requirements and verifications (See Fig. 4.1), (3) establish traceability and capture allocations to hardware, software, interfaces, and humans, and (4) detect, identify, quantify, and direct attention to scope or requirements creep.

Requirements must be under configuration control. Commercial requirements tools such as DOORS and RequisitePro help perform this function. When a defect report is written and a change in requirements is suggested as a solution, the suggested change must be submitted to a change control board. If the change control board approves of the change, then the requirements are updated at the next baseline change.

4.6.13 Verify and Validate Requirements

Verification and validation are not synonyms! Because these terms are often confused, let us examine the following dictionary definitions:

> **verify**, *v.t.* **1**. to prove to be true by demonstration, evidence or testimony; confirm; substantiate. **2**. to test or check the accuracy or correctness of, as by investigation, comparison with a standard, or reference to the facts.

validate, *v.t.* to prove to be valid.

valid, *adj*. **1**. having legal force; properly executed and binding under law. **2**. well-grounded on principles or evidence; able to withstand criticism or objection, as an argument; sound. **3**. effective, effectual, cogent. **4**. logic correctly derived or inferred according to the rules of logic. Valid applies to that which cannot be objected to because it conforms to law, logic, the facts, etc.

Verify means to prove that it is true by testing. Validate means to prove that it conforms to law, logic, and the facts, that it is sound, effective, and cogent.

Validating a System. Building the *right system*: making sure that the system does what it is supposed to do. Validation determines the correctness and completeness of the product and ensures that the system will satisfy the actual needs of the customer.

Verifying a System. Building the *system right*: ensuring that the system complies with its requirements and conforms to its design.

Validating Requirements. Ensuring that the *set* of requirements is correct, complete, and consistent, that a model can be created that satisfies the requirements, that a real-world solution can be built that satisfies the requirements, and that it can be verified that such a system satisfies its requirements. If systems engineering discovers that the customer has requested a perpetual motion machine, the project should be stopped.

Verifying Requirements. Proving that each requirement has been satisfied. Verification can be done by logical argument, inspection, modeling, simulation, analysis, test, or demonstration.

Verification and Validation. MIL-STD-1521B (and most systems engineers) and DoD-STD-2167A (and most software engineers) used the words verification and validation in almost the opposite fashion (Grady, 1994). To add further confusion, ISO-9000 tells you to verify that a design meets the requirements and to validate that the product meets requirements. NASA has a different spin. It says that verification consists of proving that a system (or a subsystem) complies with its requirements, whereas validation consists of proving that the total system accomplishes its purpose (Shishko and Chamberlain, 1995). In the Capability Maturity Model Integration (CMMI), validation (VAL) is system validation, verification (VER) is requirements verification, and requirements development (RD) Specific Goal 3 covers requirements validation. So, at the beginning of the system development project, it is necessary that the project team and customer representatives all agree on the definitions of verification and validation.

4.6.13.1 System Validation and Verification System validation artifacts can be collected at the following discrete events: white papers, trade-off studies, phase reviews, life-cycle reviews, red team reviews, SRR, PDR, CDR, and field test.

System validation artifacts that can be collected continuously throughout the life cycle include results of modeling and simulation and a count of the number of operational scenarios (use cases) modeled. Detectable system validation defects include mismatches between the model/simulation and the real system.

A sensitivity analysis can reveal validation errors. If a system is very sensitive to parameters over which the customer has no control, then it may be the wrong system for that customer. If the sensitivity analysis reveals the most important parameter and that result is a surprise, then it may be the wrong system. If a system is more sensitive to its parameters than to its inputs, then it may be the wrong system or the wrong operating point. If the sensitivities of the model are different from the sensitivities of the physical system, then it may be the wrong model.

Validation defects can be detected at inspections: the role of Tester should be given an additional responsibility—validation. Tester should read the vision statement and the concept of operation and specifically look for such system validation problems.

Systems are primarily verified at Total System Test. Sensitivity analyses can also be used to help verify systems. In a manmade system or a simulation, unexpected excessive sensitivity to any parameter is a verification mistake. Sensitivity to interactions should definitely be flagged and studied: such interactions may be unexpected and undesirable.

4.6.13.2 Requirements Validation and Verification Validating requirements means ensuring that the *set* of requirements is correct, complete, and consistent, a model that satisfies the requirements can be created, and a real-world solution can be built and tested to prove that it satisfies the requirements. If the requirements specify a perpetual motion machine, the project should be stopped.

Here is an example of an invalid requirements set for an electric water heater controller:

$$\begin{aligned}
&\mathit{If}\ 70^\circ < \mathit{Temp} < 100^\circ,\ \mathit{then\ output}\ 3000\ \mathit{watts}\\
&\mathit{If}\ 100^\circ < \mathit{Temp} < 130^\circ,\ \mathit{then\ output}\ 2000\ \mathit{watts}\\
&\mathit{If}\ 120^\circ < \mathit{Temp} < 150^\circ,\ \mathit{then\ output}\ 1000\ \mathit{watts}\\
&\mathit{If}\ 150^\circ < \mathit{Temp},\ \mathit{then\ output}\ 0\ \mathit{watts}
\end{aligned}$$

This set of requirements is incomplete: What should happen if Temp $< 70^\circ$? They are inconsistent: What should happen if Temp $= 125^\circ$? They are ambiguous, because units are not given: Are those temperatures in degrees Fahrenheit or Centigrade?

Detectable requirements validation defects include incomplete or inconsistent sets of requirements or use cases, anything that does not trace to top-level customer requirements (vision), and test cases that do not trace to use cases scenarios.

Validation defects can be detected at inspections: the role of Tester should be given an additional responsibility—validation. Tester should read the vision statement and concept of operations (ConOps) and specifically look for such requirements validation problems.

Each requirement is verifiable by (ordered by increasing cost) logical argument, inspection, modeling, simulation, analysis, test, or demonstration. Here are dictionary definitions for these terms.

Logical argument: A series of logical deductions.

Inspection: To examine carefully and critically, especially for flaws.

Modeling: A simplified representation of some aspect of a system.

Simulation: Execution of a model, usually with a computer program.

Analysis: A series of logical deductions using mathematics and models.

Test: Applying inputs and measuring outputs under controlled conditions (laboratory environment).

Demonstration: To show by experiment or practical application (flight or road test). Some sources say demonstration is less quantitative than test. Demonstrations can be performed on electronic breadboards, plastic models, stereolithography models, prototypes made in the laboratory by technicians, preproduction hardware made in the plant using developmental tooling and processes, and production hardware using full plant tooling and production processes.

Here are some examples of requirements verification. “The probability of receiving an incorrect bit on the telecommunications channel shall be less than 10^{-3}.” This requirement can be verified by laboratory tests or demonstration on a real system. “The probability of loss of life on a manned mission to Mars shall be less than 10^{-3}.” This certainly is a reasonable requirement, but it cannot be verified through test. It might be possible to verify this requirement with analysis and simulation. “The probability of the system being canceled by politicians shall be less than 10^{-3}.” Although this may be a good requirement, it cannot be verified with normal engineering test or analysis. It might be possible to verify this requirement with logical arguments.

The *Hammer of Verification* was used to verify that a pope was dead and not just sleeping. The Cardinal Chamberlain whacked the corpse on the forehead with the hammer. This verification technique was last used on Leo XIII in 1903.

There is a trade-off between the cost of verifying a requirement and the value that verification adds. Good enough is often a forgotten concept in our rush to six sigma. Creative people can always see another feature or a better way. There is a time to send the creativity to the next project and get on with the implementation of the current project. We have seen many examples where enormous amounts of money were spent developing complex computer models for verification that predicted system performance criteria to several significant digits. Whereas a reasonable prototype and simple measurements demonstrated that the performance requirement was exceeded by plus 25% or minus 15%. Usually we only need to demonstrate that the requirement is met or exceeded. We do not have to determine the exact level to high accuracy.

Traditional requirements verification is done by the test organization during the testing phase. However, this is costly and ineffective for dealing with the scale and complexity of modern systems. Built-in self-test and automated regression testing are better alternatives.

4.6.14 Manage Risk

Identifying and mitigating project risk is the responsibility of management at all levels in the company (Bahill and Karnavas, 2000; Smith and Bahill, 2007). Each item that poses a threat to the cost, schedule, or performance of the project must be identified and tracked. The following information should be recorded for each identified risk: name, description, type, origin, probability, severity, impact, identification number, identification date, work breakdown structure element number, risk mitigation plan, responsible team, needed resolution date, closure criteria, principal engineer, status, date, signature of team leader. Forms useful in identifying and mitigating risk are given in Chapter 17 of Kerzner (1995), Section 4.10 of Grady (1993), and Chapter 3 of this handbook. For the solar oven project, we identified the following risks:

1. Insufficient internal oven temperature was a performance risk. Its origin was Design and Manufacturing. It had high probability and high severity. We mitigated it by making it a technical performance measure, as shown in Figure 4.8.
2. High cost of the oven was a cost risk. Its origin was the Design process. Its probability was low, and its severity was medium. We mitigated it by computing the cost for every design.
3. Failure to have an oven ready for testing posed a schedule risk. Its origin was Design and Manufacturing. Its probability was low, but its severity was very high. We mitigated this risk by requiring final designs 7 days before the scheduled test date and a preproduction unit 3 days in advance.

Models (or computer simulations) are often used to reduce risk. Low-risk portions of the system should be modeled at a high level of abstraction, whereas high-risk portions should be modeled with fine resolution.

4.6.15 Manage the Requirements Process

The requirements process of Figure 4.6 should be tailored for each project. This tailoring will accommodate special schedule and cost constraints of the project and will incorporate customer specific needs. All of these changes should be studied to see if they would yield improvements in the company's standard requirements process. This task of continually monitoring a process, in order to make changes to improve the process, is arguably the most important task of any process. But this task is often forgotten when the process is designed.

4.7 FITTING THE REQUIREMENTS PROCESS INTO THE SYSTEMS ENGINEERING PROCESS

The Requirements Discovery Process of Figure 4.6 is one subprocess of the Systems Design Process shown in Figure 4.10.

Systems engineering is a fractal process. (This metaphor was created by Bill Nickell at Sandia Laboratories in 1994.) There is a vertical hierarchy. It is applied at levels of greater and greater detail (Bahill et al., 2008): it is applied to the system, then to the subsystems, then to the components, and so on. It is applied to the system being designed, to the enterprise in which the system will operate, and to the enterprise developing the system. The enterprise developing the system grows and expands as the system development process matures. Then the system development enterprise goes through a metamorphosis as the process moves into the production and verification phases. The system development enterprise is not a static organization, rather it is a living organism that grows and undergoes dramatic changes during system development. The systems engineering process is also applied horizontally. It is applied to alternative 1, then to alternative 2, then to alternative 3, and so on. It is applied to component 1, component 2, component 3, and so on. This process is recursive and iterative, and much of it is done in parallel. This concept is shown in a poster that is available at http://www.sie.arizona.edu/sysengr/fractal.gif.

The fact that Discover Requirements (Fig. 4.6) is a subprocess in the System Design Process (Fig. 4.10) and that the System Design Process is a subprocess in the Systems

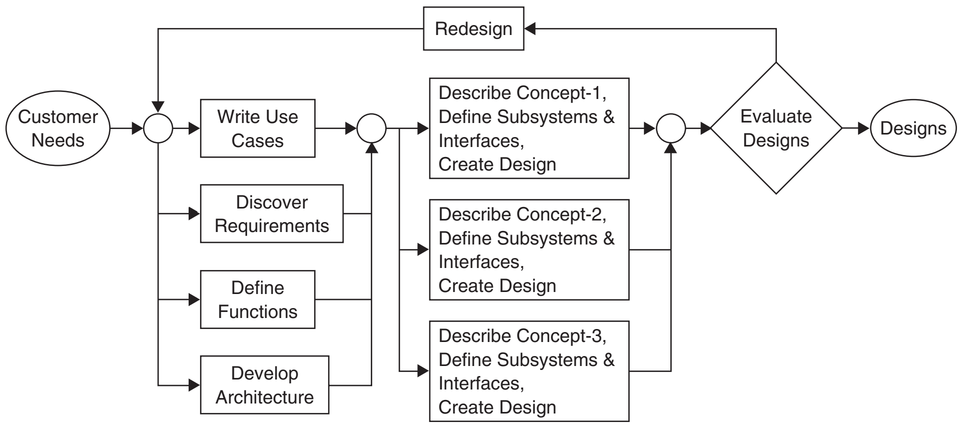
Redesign
Customer Needs
Write Use Cases
Discover Requirements
Define Functions
Develop Architecture
Describe Concept-1, Define Subsystems & Interfaces, Create Design
Describe Concept-2, Define Subsystems & Interfaces, Create Design
Describe Concept-3, Define Subsystems & Interfaces, Create Design
Evaluate Designs
Designs

Figure 4.10 The system design process tailored for the preliminary design phase. (Copyright © 2002, A. T. Bahill, from http://www/sie.arizona.edu/sysengr/slides/. Used with permission.)

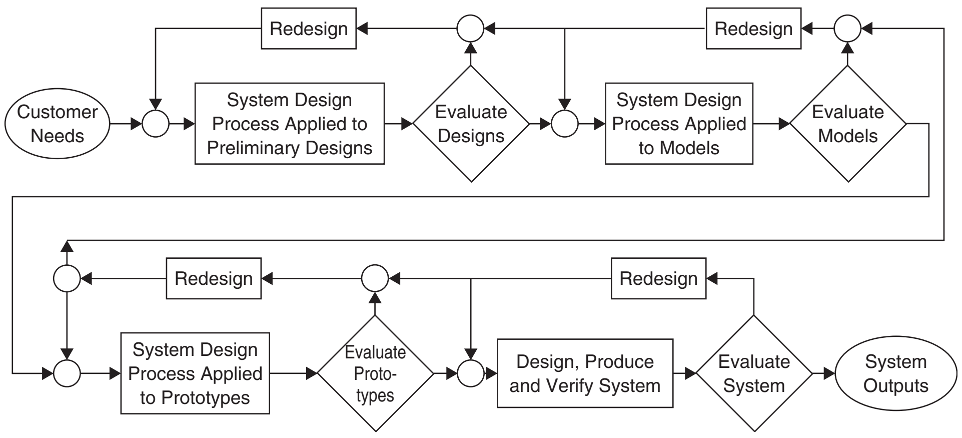
Redesign
Redesign
Customer Needs
System Design Process Applied to Preliminary Designs
Evaluate Designs
System Design Process Applied to Models
Evaluate Models
Redesign
Redesign
System Design Process Applied to Prototypes
Evaluate Proto-types
Design, Produce and Verify System
Evaluate System
System Outputs

Figure 4.11 The design portion of the systems engineering process.

Engineering Process of Figure 4.11 illustrates the hierarchical nature of systems engineering. We will now try to explain the repetitive aspect of the Systems Engineering Process. In Figure 4.11, the System Design Process is applied to preliminary designs, models, and prototypes and to the real system. However, this process is not a waterfall. Each of the loops will be executed many times. Execution of the Redesign loop in the upper left is very inexpensive and should be exercised often. Execution of the Redesign loop in the lower right is expensive but should be exercised. Whereas, execution of the Redesign loop from the lower right all the way back to the upper left is very expensive and should seldom be exercised.

Figure 4.11, of course, only shows a part of the Systems Engineering Process. It omits logistics, maintenance, risk management, reliability, project management, documentation, and so on (Chapman et al., 1996: Martin, 1996: Rechtin and Maier, 1996).

Although it seems that the Systems Design Process has been applied like a cookie cutter to the four phases—preliminary designs, models, prototypes, and the real system—the emphasis should be different in each application. In the preliminary design phase, the emphasis should be on discovering requirements and defining functions, with some effort devoted to alternative concepts. In the modeling phase, the emphasis should be on describing, analyzing, and evaluating alternative concepts, with some effort devoted to rewriting requirements and redefining functions. In the prototype phase, the emphasis should be on evaluating the prototype, modeling the highest cost or highest risk components, with some effort devoted to rewriting requirements, redefining functions, (and if there are multiple prototypes) describing alternatives, and analyzing alternative concepts. And finally, when applied to the real system, the emphasis should be on evaluating the system. When the System Design Process is applied in the four phases, the inputs and outputs are unique and different for each phase and must be carefully labeled accordingly. For the preliminary design phase, the input is the customer request and the output is the candidate designs. For the modeling phase, the input is candidate designs and the output is models and modeling results that are the inputs for .... the prototype phase, and so on.

4.8 RELATED ITEMS

1. *Requirements Have Attributes.* The following are common attributes of requirements: priority, rationale, status, risk level, owner, cost, revision, date requested, associated TPM, effort, date approved, difficulty, trace to (req), stability, trace from (req), origin, and assigned to (person).

2. *Requirements on Whom?* Requirements usually describe or constrain the system being designed. In designing an ATM machine we might write "The ATM machine shall dispense cash within 10 seconds." But this might impose a requirement on the central bank: "The central bank shall confirm the account and balance within 5 seconds of request." This requires the definition of system boundaries. The importance of this is that requirements on other entities are not requirements on your system. Statements like "The pilot shall ..." are operation statements, not requirements (Hooks and Farry, 2001). Having identified the target for each requirement, gather all of the requirements that are not requirements on your system and move them to a separate document.

3. *Product or Process.* Requirements can describe the product or the process for producing the product (Abadi and Bahill, 2003). Product configuration items include product processing, system management, framework, database, catalog processing, hardware, and software. Example process requirements include the following

"DOORS shall be used to manage requirements."

"IBM Rational Rose shall be used for design."

"The project shall hold the phase reviews described in a certain document."

UML support tools like Rose and Enterprise Architect call the process documents the Business Model.

4. *Requirements Versus Constraints.* A requirement allows more than one design option (trade-off requirements). A constraint leaves no options (mandatory requirements). A mandated algorithm, database, or technology would be a constraint.

5. *High-Level Requirements.* The system's high-level requirements are contained in the Mission Statement, the Concept of Operations, and the Business Model. But we should not call them requirements. They lack the formality of requirements. They

should be called goals or capabilities. Then the term requirements can be reserved for formal requirements like the ones that are typically in a requirements database. The high-level documents describe the environment the system will operate in and the architecture of the system.

6. *Requirements Versus Goals.* The Mission Statement, Concept of Operations, and Business Model contain goals, objectives, features, and capabilities. Formal requirements are contained in the Specific Requirements Sections of the Use Cases, Supplementary Requirements Specification, and Tradition Requirements Specification (e.g., a DOORS requirements database).

7. *Concept of Operations Versus Operation Concept Description.* The Concept of Operations (ConOps) describes the mission of an enterprise and its component systems. It gives a broad outline describing a series of operations that the systems will go through. It states what the systems must do and how they fit into the enterprise. It may include business process models. It is usually supplied by the government or the customer. The Operational Concept Description (OCD) is a lower-level description of how an individual system is to be used. It is usually written by the contractor. It is a Contract Deliverable Requirements List (CDRL) validation item. It contains text, uses cases, and diagrams. But when the prime contractor's Operational Concept Description is given to a subcontractor, it becomes the subcontractor's Concept of Operations.

8. *External Versus Internal.* Some engineers characterize requirements as external and internal. External requirements are driven by customer need; internal requirements are driven by company practices and resources. For example, a systems development company might require certain processes or technologies be used in meeting customer requirements. Some customers will selct a supplier/system developer based on the supplier's internal processes and controls. The customer has confidence the supplier's processes and controls will deliver a quality product.

9. *Requirements Versus Specifications.* A requirements definition set, which we usually call the requirements, describes the functions the systems should provide, the constraints under which it must operate, and the rationale for the requirements. It should be written in plain language. It is intended to describe the proposed system to both the customer and the designers. It should be broad so that many alternative designs fit into its feasibility space.

The requirements specification, which we usually call the specification or the spec, provides a precise description of the system that is to be built. It should have a formal format and might be written in a technical language. It is intended to serve as the basis of a contract between Purchasing and Manufacturing. It should narrow the feasibility space to a few points that describe the system to be manufactured.

The set of requirements determines the boundaries of the solution space. The specifications define a few solutions within that space. The requirements say what; the specifications say how.

10. *Grouping of Requirements.* Requirements should be organized into categories, subcategories, and so forth. Requirements that are correlated should be grouped together. Suppose a young couple wants to buy a new car. The man says his most important requirement is horsepower and the woman says her most important requirement is gas mileage. Although these are conflicting requirements, with a negative correlation, there is no problem. Their decision of what car to buy will

probably be based on a trade-off between these two requirements. Now, however, assume there is another couple where the woman says her only requirement is safety (as measured by safety claims in advertisements), but the man says his most important requirements are lots of horsepower, lots of torque, low time to accelerate 0 to 60 mph, low time to accelerate 0 to 100 mph, low time for the standing quarter mile, large engine size (in liters), and many cylinders. Assume the man agrees that the woman's requirement is more important than his. So they give safety the maximum importance value of 10, and they only give his requirements importance values of 3 and 4. What kind of a car do you think they will buy? The man's requirements should have been grouped into one subcategory, and this subcategory should have been traded off with the woman's requirement. In summary, similar, but perhaps dependent, requirements ought to be grouped together into subcategories.

11. *Requirements Hierarchy.* Grouping the requirements into categories helps to put the requirements into a hierarchy. The requirements database should have a tree-like structure. It should link parent–child relationships and identify siblings. It is very important that requirements at different levels be treated separately (Bahill et al., 2008).

4.9 REQUIREMENTS VOLATILITY

It is not possible to get all of the requirements correctly, up front. Requirements are emergent. Customers believe in the "I'll know it when I see it" maxim. Customers' priorities change with time. Once low-level requirements are satisfied, then other requirements become high priority. However, requirements volatility per se is not bad: requirements volatility is a good thing. It means you are doing your job of deriving, clarifying, and evolving requirements.

Throughout the design process the requirements will change. Don't whine or complain about it. Design your system so that changing requirements will affect as few subsystems as possible. The following principles (from Bahill and Botta, 2008) will help with changing requirements:

- State what not how.
- Use hierarchical, top–down design.
- Work on high-risk items first.
- Use evolutionary development.
- Understand your enterprise.
- Employ rapid prototyping.
- Develop iteratively and test immediately.
- Create modules.
- Create libraries of reusable objects.
- Use open standards.
- Design the interfaces.
- Control the level of interacting entities.
- Identify things that are likely to change.
- Write extension points.
- Group data and behavior.

- Use data hiding.
- Do not allow undocumented functions.
- List functional requirements in the use cases.
- Allocate each function to only one component.
- Write a glossary of relevant terms.
- Provide observable states.
- Produce satisficing designs.
- Do not optimize early.
- Maintain an updated model of the system.
- Develop stable intermediates.
- Design for testability.
- Envelope requirements.
- Create design margins.
- Design for evolvability.
- Build in preparation for buying.
- Create a new design process.
- Change the behavior of people.

Traceability helps manage change. Business needs drive customer needs that produce user needs that demand system features that engineers implement, test, and document (See Fig. 4.1). Such tracing helps ensure that when a requirement is changed, related requirements also reflect change. For example, if the customer no longer needs a solution for a particular part of a problem, then certain features, requirements, and tests can be eliminated.

Using requirements traceability will help you to (1) verify that the system does what it is supposed to do, (2) ensure that the system does only what it is supposed to do, (3) assess impact of change, (4) find related requirements, and (5) test related requirements. The basic strategy is to trace system requirements into derived requirements, then trace requirements into designs, then trace requirements into test procedures, and finally trace requirements into documented plans.

4.10 INSPECTIONS

To detect mistakes, software code, use cases, and requirements are now being regularly inspected. Several engineers are each given 250 lines of code or text to study. (The exact number of lines varies with the type of material and the phase in the system life cycle.) Each inspector records how much time he/she spent and how many lines he/she inspected. Each inspection meeting lasts 2 hours. At the inspection there is a chair, a recorder, the author, and several inspectors. Each inspector points out mistakes he/she found. The criticality of each mistake is determined and it is assigned to a particular person for action. Formal inspections such as these provide a wealth of data for the metrics analysis group. Inspections save time and money and increase performance. Each defect caught at an inspection takes, on average, 3 person-hours to correct. Each defect caught in integration takes about 20 person-hours to correct. Each defect caught postdelivery takes about 40 person-hours to correct.

4.11 A HEURISTIC EXAMPLE OF REQUIREMENTS

Earlier, we discussed several ways to express requirements, such as narratives, shall and should statements, and computer models. Here is another example, one that uses formal logic notation. LaPlue et al. (1995) state that a requirement should contain (1) the description of a system function and its output, (2) the name of the system that accepts this output, (3) conditions under which the requirement must be met, (4) external inputs associated with the requirement, and (5) all conditions that determine if the system output is correct. The authors have organized this into a standard template:

```
The system shall ⟨function⟩
  for use by ⟨users⟩,
  if ⟨conditions⟩,
  using ⟨inputs⟩,
  where ⟨conditions⟩.
```

The ⟨function⟩ is usually of the form ⟨verb⟩ ⟨output⟩

They offer the following example.

Requirements for an Automated Teller Machine

3.0 Transaction Requirements

3.1 Related to the ATM User

- 3.1.1 Produce Receipt
- 3.1.2 Dispense Cash

  The ATM shall dispense cash
  - for use by the ATM user
  - if the ATM user requested a withdrawal
  - and if the Central Bank verified the account and PIN
  - and if the Central Bank validated the withdrawal amount
  - and if the ATM cash on hand is greater than or equal to the cash requested
  - using the Withdrawal Validation Message from the Central Bank and the Account Verification Message from the Central Bank and the Withdrawal Request from the user
  - where the amount of cash produced equals the amount requested
  - and where the cash is dispensed within 10 seconds of the receipt of the Withdrawal Validation Message from the Central Bank
- 3.1.3 Eject Card
  - 3.1.3.1 Eject bank card at end of session

    The ATM shall eject the bank card
    - for use by the ATM user
    - if the ATM user has inserted a bank card
    - and if the ATM user has requested termination of session
    - using the Bank Card and the Terminate Request
    - where the Bank Card is ejected within 1 second of the receipt of the Terminate Request

3.1.3.2 Eject unreadable cards
The ATM shall eject the bank card
- for use by the ATM user
- if the ATM user has inserted a bank card
- and if the bank card does not contain a valid code
- using the bank card
- where the code reading and validation is as specified in Bank Card Specifications, Section w.x.y

3.1.4 Produce Error Messages

3.2 Related to the Central Bank

3.2.1 Verify Account Message
The ATM shall produce the Verify Account Message
- for use by the Central Bank
- if the ATM user has entered a PIN
- and if the bank card contains a readable code
- using the bank card and user-entered PIN
- where the content and format is as specified in the Central Bank Interface Specification, Section x.y.z
- and where the message is issued within 1 second of the final digit of the PIN

This example shows many of the features of good requirements that were mentioned in this chapter. The numbering scheme manifests the tree structure of this set of requirements: parent, child, and sibling relationships are clear. References are made to the specifications. In each requirement, the customer is identified: for example, the ATM user and the Central Bank. Many sequence diagrams were used to elicit these requirements. Performance evaluation criteria are given, they are specified as maximum values, units are given, and they are testable: for example, cash must be dispensed within 10 seconds. The requirements state what, not how: for example, "The ATM shall dispense cash." The requirements identify applicable states with the conjunctive *if* clauses. The word choice is correct. It is unfortunate that there is no provision for the rationale.

4.12 THE HYBRID PROCESS FOR CAPTURING REQUIREMENTS

So far, this chapter has followed traditional industry practices for requirements. But it has not presented a process for actually capturing the requirements (Section 4.6.5 was vague). Modern system design is now *use case* based. The use cases capture the required functional behavior of the system. So then, why not use the use cases to capture the requirements? The following presentation of this hybrid process for capturing requirements is based on Daniels and Bahill (2004).

We start this section with a description of a use case and an example. A use case is an abstraction of a required behavior of a system. A use case produces an observable result of value to an actor. An actor is the role that a user (or some other object) plays with respect to the system. A typical system will have many use cases, each of which satisfies a goal of a particular actor. Each use case describes a sequence of interactions between one or more actors and the system. A use case narrative is written in a natural language. It is written as a sequence of numbered steps with a main success scenario and alternate flows. This sequence of interactions is contained in a use case report and the relationships between the system and its actors are portrayed in use case diagrams. Here is an extremely simple example of a use case for the status-monitoring portion of a heating, ventilation, and air conditioning (HVAC) system for a typical Tucson house (Bahill and Daniels, 2003).

Name: Display System Status.

Brief Description: Indicate the health of the HVAC system.

Added Value: Home Owner knows if the system is working properly.

Scope: An HVAC controller for a Tucson house.

Primary Actor: Home Owner.

Frequency: Typically once a day.

Main Success Scenario

1. Home Owner asks the system for its status.
2. Each component in the system reports its status to the Controller.
3. The system accepts this information and updates the System Status display.
4. Home Owner observes the System Status [exit use case].

Alternate Flows

Specific Requirements

Functional Requirements

1. Each component shall report its status to the Controller.
2. The system shall display the System Status.

Nonfunctional Requirement: The System Status shall be displayed within 2 seconds of request.

Author: Terry Bahill.

Last Changed: June 20, 2007.

Use cases can be used to derive traditional requirements. A traditional requirements specification, which provides imperative textual statements, serves as the primary means for systems engineers to bound and communicate system capabilities and constraints. However, in recent years the popularity of documenting software requirements with use cases has increased dramatically, with use case-based analysis becoming the foundation of modern software analysis techniques. Due to the success of the use case approach in the software community, systems engineers have started using use cases as a tool to document system requirements (Bahill and Daniels, 2003). Using a common methodology for requirements elicitation helps align systems engineers and software engineers, allowing for an easier transition from system and subsystem requirements into software requirements. We think that the combination of shall-statement requirements and use case modeling techniques provides a complementary and synergistic way to document and communicate relevant information to effectively design, develop, and manage complex systems. We believe that the resultant quality and clarity of knowledge gained through the combination of two methodologies, use cases and shall-statement analysis, is worth the time spent.

This section assumes a rudimentary knowledge of use cases. Many references are available to introduce unfamiliar readers to use cases (Armour and Miller, 2001; Bahill and Daniels, 2003; Cockburn, 2001; Daniels and Bahill, 2004; Kulak and Guiney, 2000; Leffingwell and Widrig, 2000; Övergaard and Palmkvist, 2005).

4.12.1 Why Write a Requirements Specification?

Requirements are documented so that the characteristics of the to-be system can be discussed and analyzed by engineers and stakeholders to facilitate a shared clear vision of the proposed system. Systems engineers and architects use the requirements set to ensure system and enterprise-wide coverage of capabilities and characteristics. Implementation engineers use the agreed-upon requirements set to derive additional requirements and develop subcomponents that contribute to a system that meets those requirements. Program managers track requirements volatility to measure and monitor risks associated with requirements creep and redefinition. Customers sign off on requirements during acceptance testing to ensure that the capabilities offered by the system meet their needs and intent. It is clear that gathering, discussing, and deriving requirements is an essential step in the engineering process to ensure that the system, once built, actually satisfies the stakeholders' needs.

4.12.2 Problems with Traditional Requirements

A traditional requirements specification attempts to document the imperative functionality and constraints of a system by enumerating each requirement using "shall" notation through which individual capabilities and constraints are expressed (e.g., "The system shall . . ."). The requirements set is typically structured according to functional areas, and each requirement is attributed, possibly traced to and traced from other requirements. Shall statements are used to describe different types of requirements such as functional, performance, security, reliability, availability, and usability.

However, if the requirements specification consists solely of shall-statement requirements, then for even a moderately complex system, the requirements set can become unwieldy, containing hundreds or thousands of requirements. This by itself is not a problem, as modern systems are very complex and often need this level of detail to avoid ambiguity. The problem is that documenting the requirements as a set of discrete and disembodied shall statements without context makes it difficult for the human mind to comprehend the set and fully interpret the intent and dependencies of each requirement. This makes it hard to detect redundancies and inconsistencies. In short, large shall-statement requirement specifications make it difficult for engineers and customers to really understand what the system does! Because there is no straightforward way to assimilate the requirements set, requirements are often misinterpreted, redundant, and incomplete (Bahill and Henderson, 2005), which can result in inferior, overly expensive systems and ultimately dissatisfied stakeholders.

4.12.3 Use Cases

Use cases have been accepted by the software community as a requirements gathering and documentation tool that captures system requirements through generalized, structured scenarios that convey how the system operates to provide value to at least one of the system's actors. The primary reason why use cases have become a popular method is the simple and intuitive way in which the system's behavior is described. Use case models are designed to serve as a bridge between stakeholders and the technical community. Through a use case model, stakeholders should be able to readily comprehend how the system helps them fulfill their goals. Simultaneously, engineers should be able to use the same information as a basis for designing and implementing a system that executes the use cases.

A set of use cases (collectively referred to as the use case model) should describe the fundamental value-added services that users and other stakeholders need from the system (Adolph and Bramble, 2003). Use cases can be thought of as a structured, scenario-based method to develop and represent the behavioral requirements for a system. The use case approach subscribes to the notion that each system is built to support its environment or actors—be it human users or other systems. Use cases, by definition, describe a series of events that, when completed, yield an observable result of value to a particular actor (Jacobson et al., 1995). The fundamental concept is that systems designed and developed from use cases will better support their users.

Ultimately, use cases have been shown to be a simple and effective tool for specifying the behavior of complex systems (Jacobson et al., 1995). Therefore, it is no surprise that systems engineers have started using this method for documenting and communicating requirements.

4.12.4 The Difficulty

Although use cases solve some of the problems with specifying requirements when contrasted with the shall-statement method, there are issues with applying use cases to solve systems engineering problems. There are a few possible reasons for these issues:

1. Use cases don't *look* like a traditional requirements specification.
2. Use cases *feel* somewhat vague.
3. Use cases do not contain *all* of the requirements.
4. The completeness of a set of use cases is sometimes difficult to assess, especially for unprecedented complex systems.

Other contributing factors result because older systems engineers and customers do not have much experience with use cases. Systems engineers have been accustomed to developing traditional requirement specifications. Use cases were a departure from the past. It is our belief that use cases will soon be commonplace. Until then, it is important to educate customers and systems engineers on the use case concepts and the fact that employing use cases into the overall systems engineering process will help reduce risk and increase the probability of delivering the system expected by the customer.

Additionally, the UML specifications (Object Management Group, 2007) do not provide much guidance on applying use cases. Many authors, including Cockburn (2001), Armour and Miller (2001), and Bahill and Daniels (2003), have improved on the available literature and give good practical direction on applying use cases. The IBM Rational Unified Process provides detailed guidance on practical methods to employ use cases in both systems and software engineering projects (Jacobson et al., 1999).

4.12.5 Strengths and Weaknesses

Use case modeling has shown strengths in many areas of requirements capture. The dialogue between the system and the actor that is expressed in a use case's sequence of events explains clearly how a system reacts to stimuli received by the system's actors. What engineers typically do not realize is that the actual functional requirements are embedded in the sequence of events. In other words, a use case describes the requirements. Whenever a use case's sequence of events indicates that the system performs some function, a functional requirement has been imposed on the system. Each use case is chock full of functional requirements (possibly multiple requirements per sentence!) when viewed this way. By design, the requirements are an integral part of a use case's natural language story.

A use case's strength is also its weakness. To keep use cases simple, readable, and manageable, they can only tell a fraction of the complete story without becoming unwieldy and difficult to understand (Cockburn, 2001). Use cases alone were not meant to capture all of the requirements. A use case is very good at capturing the functional requirements for a system in an understandable and unthreatening way. However, shall statements, diagrams, tables, equations, graphs, pictures, pseudocode, state machine diagrams, sequence diagrams, use case diagrams, statistics, or other methods must still be used to capture additional requirements and add richness to provide a sufficient level of detail to adequately characterize a system. All of these artifacts may be related to a use case, but they are typically not expressed directly into a use case's natural language story. In some instances, use cases are not the best method to express a system's functional requirements. For example, systems that are algorithmically intense and do not have much interaction with their environment may be better described using some other method—although use cases can be used in these cases too (Cockburn, 2001; Jacobson, 2000).

To summarize, use cases excel on being understandable, at the expense of being potentially ambiguous. Shall-statement requirements are typically very well defined and often stand alone, where individual capabilities are expressed in a rather abstract way, out of context with the rest of the system's characteristics. The use of shall-statement requirements alone makes it more difficult for engineers and stakeholders to assess one requirement in a virtual sea of disjointed capabilities. Shall statements are very good at precise expression, while use cases are good at communicating requirements in context. This tension promotes the need for a combined use case–shall-statement approach, where the system's behavior is documented precisely and understandably in the same way that it is derived—by analyzing all of the discrete, end-to-end interactions between actor and system that ultimately provide some value to an actor.

4.12.6 Hybrid Requirements Capture Process

The Specific Requirements section of a use case report was originally intended to capture the nonfunctional requirements, which are typically performance requirements on a particular use case step, using shall-statement notation. Daniels and Bahill (2004) proposed using the Specific Requirements section to capture not only nonfunctional but also functional requirements specified in the use case using shall-statement notation. When the shall-statement requirements are captured in this way, they retain their context since they can readily be traced to the use case sequence of events from which they were derived. Using this approach, the understandability of the use case is balanced by the razor-sharp precision of shall statements.

Engineers will also discover requirements that do not fit nicely within the context of one use case, but rather they characterize the system in general. These global requirements should be put into a separate Supplementary Requirements Specification using shall-statement notation. The Supplementary Requirements Specification is independent of the use case model and is intended to contain all of those requirements that do not apply cleanly to any single use case. Requirements that describe physical constraints, security (physical security, antitampering and information assurance), quality, reliability, safety, usability, supportability requirements, or adherence to standards, for example, are good candidates for inclusion in the Supplementary Requirements Specification. These types of requirements are typically not local to a specific use case. The hybrid process described in this section includes the use of a Supplementary Requirements Specification for capturing these classes of requirements.

The use case model (which contains the individual use case reports) together with the Supplementary Requirements Specification constitutes a way to completely document a system's requirements in an exact, yet understandable manner. In many cases, this alone will be satisfactory for specifying the requirements. However, some customers may still dictate that a traditional shall-statement specification be developed. We discuss this in the next section.

Table 4.1 uses some of the qualities of good requirements to compare and contrast the three requirement-gathering methods presented so far: shall statements, use cases, and the hybrid process using both use cases and shall statements. Each technique is given a rating from 1 to 3 for each quality (3 being the best) to assess how well the technique promotes requirements best practices. We believe that the hybrid process encompasses the strengths of each technique and excels in each quality.

Necessary. The use case method excels here. Since use cases are driven by actor needs, each use case represents a set of capabilities that are of value to the system's stakeholders. It is hard to determine whether a shall-statement requirement is necessary without considering the entire requirements set and additional documentation. The hybrid process, due to its use case-driven foundation, rates highly.

Verifiable. Since shall-statement requirements are concise and discrete, they are, in principle, easier to verify than a use case that can dish out multiple requirements in a single sentence. The hybrid process retains the discrete nature of the shall-statement requirements, which makes them easier to verify.

TABLE 4.1. Rating the Requirements Methods

Quality	Shall-Statement Method	Use Cases	Hybrid Process
Necessary	1	3	3
Verifiable	3	1	3
Unambiguous	3	2	3
Complete	2	3	3
Consistent	1	2	3
Traceable	2	2	3

Unambiguous. Shall-statement requirements are more likely to be unambiguous, because they can be written much more tersely and precisely. Use cases, due to their narrative format, are not as well suited to the same accuracy in expression. The context provided by a use case helps remove some of the ambiguity. The hybrid process takes the best of both worlds—context with use cases, and freedom to use more precise wording with shall statements.

Complete. A use case is structured in terms of basic and alternative paths, giving a clear understanding of when a requirement applies, and when it does not. Since use cases are generated by considering the needs of all of the system's actors, it is much less likely that requirements will be missed. Use cases are an excellent way to help ensure that the requirements set is complete. It is difficult to determine whether a shall-statement requirement set is complete without referencing the entire documentation. But many times, due to the structure of the requirements set, you can look for incompleteness (Davis and Buchanan, 1984). The hybrid process retains the use case method's excellence in complete requirements description.

Consistent. Use cases typically deal with one goal at a time and therefore it is easier to separate requirements and reduce the risk of conflicting with other requirements. However, neither method alone excels at ensuring a consistent requirements set. The hybrid process, using the supplementary specifications, provides a mechanism to extract requirements that apply across use cases and specify them in one place. This helps maintain the consistency across the set and reduces duplicity. We do admit that requirements extracted from use case narratives may need to be refined and aggregated into a comprehensive, nonredundant set of system functional requirements for formal, high-ceremony projects.

Traceable. Use cases, through their actor-driven derivation, are easily traced to higher-level actor goals. However, it is not as straightforward to allocate specific use case requirements to the system components. Shall statements can be traced through a numbering scheme as parent–child requirements. The hybrid process retains the traceability to actor goals through use cases and allows the shall statements to be allocated to system components. The use of object models such as sequence diagrams in the hybrid specification helps allocate responsibilities and requirements to the components of the system.

4.12.7 Hybrid Process Defined

There are many methods for gathering requirements. For complex systems, we believe that a combined use case and shall-statement approach should be employed to capture a system's requirements. Use cases provide understandability, context, and direct traceability to actor needs and interfaces. Shall-statement requirements add the precision necessary to completely and unambiguously specify the system. With any approach, early and frequent interview and review iterations should be conducted with the stakeholder community to ensure that their concerns and desires are addressed.

The hybrid process for capturing requirements proceeds as follows.

1. Using the system vision statement, the concept of operations (ConOps), description of needed system capabilities, and interviews with the customer, develop a

business model to understand how the system under design fits into the overall enterprise. The business model provides context for the system and can profoundly increase the quality of later analysis. Jacobson et al. (1995) presents a practical methodology for defining business models using use case and object-oriented techniques.

2. Discover the system's actors and their goals with respect to the system under design. The business model is a valuable input in accomplishing this task. Define the system architecture.
3. Use the actor goals to sketch out the use case reports, concentrating on use case names and brief descriptions.
4. Iterate on the important or architecturally significant use cases, filling out more and more detail in the reports and capturing alternative sequences of events, preconditions, and postconditions.
5. Extract the functional requirements from each use case's Brief Description, Added Value, and flow of events and document them in the Specific Requirements Section of the use case report, or the Supplementary Requirements Specification if they apply to many use cases. Ensure that traceability is maintained.
6. In each use case report, document the nonfunctional requirements, typically performance, in the Specific Requirements section as they apply to each use case.
7. In parallel with steps 5 and 6, develop a Supplementary Requirements Specification to capture system-wide requirements that do not fit cleanly into individual use case reports.
8. Iterate on the use case set to ensure consistency and completeness as the program progresses.

If a traditional requirements specification (shall-statement requirements only) is to be developed, combine the requirements documented in each use case report's Specific Requirements section, along with the Supplementary Requirements Specification, to generate the traditional requirements specification. This should be primarily a simple copy and paste operation. Figure 4.12 shows this Hybrid Requirements Process. Most of this figure is derived from the standard UML use case process (Booch et al., 2005). The new task is *Extract functional requirements*, which is highlighted.

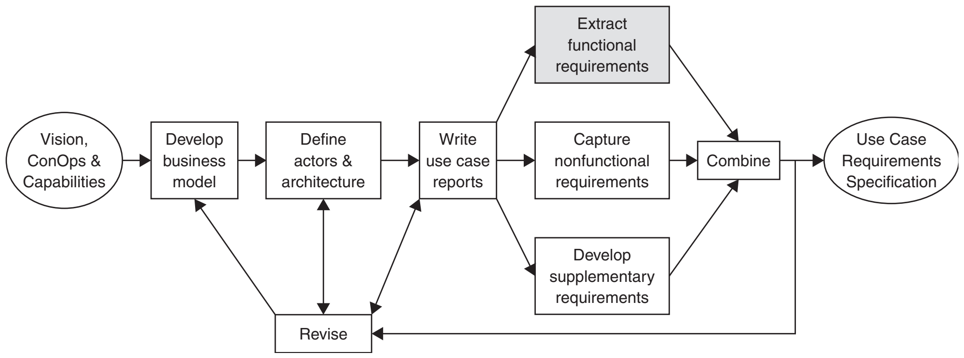
Extract functional requirements
Vision, ConOps & Capabilities
Develop business model
Define actors & architecture
Write use case reports
Capture nonfunctional requirements
Combine
Use Case Requirements Specification
Develop supplementary requirements
Revise

Figure 4.12 The hybrid process for capturing requirements in use cases. (Copyright © 2004, A. T. Bahill, from http://www/sie.arizona.edu/sysengr/slides/. Used with permission.)

Step 5 above “Extract functional requirements” is really what this section is about and therefore deserves more attention. As mentioned before, a use case describes the functional requirements. Wherever in a use case’s sequence of events, a system capability or function is called out, for example, “The system finds. . . ,” or “The system sends. . . ,” or “The system checks. . . ,” or “The system displays. . . ,” functional requirements are being imposed on the system. Step 5 above is about scanning through the use case’s sequence of events and extracting out all such statements of behavior, as well as derived requirements that are not explicitly stated. These extracted and derived requirements can then easily be translated into the shall-statement requirements, for example, “The system shall find. . . ,” “The system shall send. . . ,” “The system shall check. . . ,” “The system shall protect. . . .” This is where the precision comes in and where shall statements excel. Using natural language, as advocated with use cases to make them accessible by everyone, it is often difficult to be precise and easy to understand at the same time. Extracting functional requirements and incorporating derived requirements in this way allows for precise statements of capability without disrupting the narrative unfolding within a use case. The fact that these shall-statement requirements were extracted from and traced to use cases (very likely to a specific use case step) provides higher confidence that they actually satisfy a real need from the actor’s perspective—this is where use cases excel. Use cases also make the functional requirements derivation process straightforward. It is relatively easy to scan the use case’s text and find functional requirements, thus providing a catalyst for identifying other categories of requirements.

What we are left with is a shall-statement representation of the requirements contained in a use case, which can then be made more precise than the use case narrative. Since these requirements are traceable back to the use case narrative, we can always go back and get the context from which it was derived. We cannot stress enough that it is critical that the stakeholders be involved as much as possible in the use case generation and validation activities to ensure that the resulting specification is driven by their needs. Use cases make this easier.

Individual requirements, use cases, and the traceability and attributes of each should be managed in a requirements management tool. We recommend selecting tools that allow free-form text-based specification for documenting use cases and shall statements in addition to an underlying repository for storing and manipulating the requirements as database records. This allows use of analysis tools such as trend analysis, trace matrices, and reports, while keeping the use cases and requirements in user-friendly document format. As requirements are changed in the document, the database should automatically be kept in synch, and vice versa; if the database is updated, the document is synchronized accordingly.

The UML class diagram given in Figure 4.13 shows how use case, requirement, and specification concepts are related. For a given system, a Use Case Requirements Specification contains one Use Case Model and one Supplementary Requirements Specification. A Use Case Model contains one or more Use Case Reports. Each Use Case Report contains one or more Sequences of Events (one main sequence of events, and zero or more alternative sequences), as well as one Specific Requirements section. The sequence of events within the Use Case Report contains the informal functional requirements for that use case, while the Specific Requirements section contains the nonfunctional requirements and the formal functional requirements extracted from the Sequence of Events. The Functional Requirements trace to steps in the

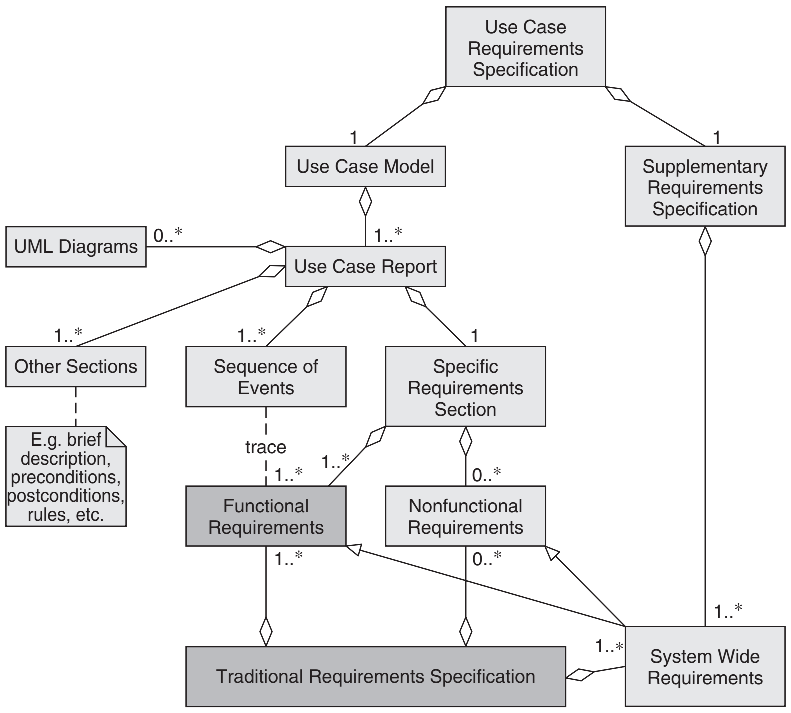
Use Case Requirements Specification
1
Use Case Model
1
Supplementary Requirements Specification
UML Diagrams
0..*
1..*
Use Case Report
1..*
Other Sections
1..*
Sequence of Events
1
Specific Requirements Section
E.g. brief description, preconditions, postconditions, rules, etc.
trace
1..*
1..*
0..*
Functional Requirements
Nonfunctional Requirements
1..*
0..*
1..*
1..*
Traditional Requirements Specification
System Wide Requirements

Figure 4.13 UML class diagram depicting the components and interrelationships of a hybrid requirements specification. Lines with arrowheads are generalizations. Lines with diamond heads are aggregation relationships. The numbers represent multiplicities. (From J. Daniels, and A. T. Bahill, (2004). Hybrid process combines traditional requirements and use cases. *Syst. Eng*. **7**(4):303–319. Copyright © 2004. Reprinted with permission of John Wiley & Sons, Inc.)

Sequence of Events. The Functional Requirements part of Figure 4.13 is the bridge between the behavior of the system and the Traditional Requirements Specification. It helps systems engineers to get the requirements right. Most of Figure 4.13 is derived from the standard UML use case process. The new parts, which are highlighted, are the *Functional Requirements* and their relationship to the *Traditional Requirements Specification*.

The Supplementary Requirements Specification contains system-wide requirements that do not fit nicely within the Sequence of Events or Specific Requirements section of one of the use cases. A Traditional Requirements Specification can be generated by combining the Functional Requirements and Nonfunctional Requirements from each Use Case Report along with the system-wide requirements from the Supplementary Requirements Specification.

In a traditional systems engineering environment, Test Procedures are detailed and sequential. Sometimes they look like a use case Sequence of Events. The Requirements Specification may have hundreds of requirements and it paints a picture of the system being designed. It is usually written before the test procedures, but it might not bear a close resemblance. The Design Model captures the architecture and the interfaces. What do these three views of the system have in common? In our hybrid process,

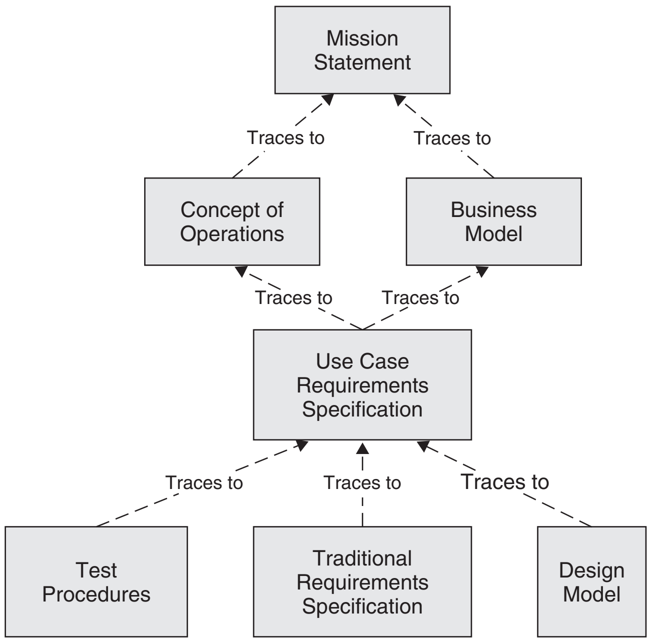

Figure 4.14 The ancestry model for system documentation. (From J. Daniels and A. T. Bahill, (2004). Hybrid process combines traditional requirements and use cases. *Syst. Eng*. **7**(4):303–319. Copyright © 2004. Reprinted with permission of John Wiley & Sons, Inc.)

they have a common ancestor: the Use Case Requirements Specification, as shown in Figure 4.14.

A human, a chimpanzee, and an orangutan have a common ancestor. Archeologists studying primate evolution have never seen this missing link, but they have a good idea what it would look like. Tracing ancestors has been a big research effort over the last few centuries: it evidentially is a useful scientific method. Continuing with this analogy, the Use Case Requirements Specification is the missing link in the traditional systems engineering process. Its place is shown in Figure 4.14.

The Mission Statement, Concept of Operations, and Business Model contain goals, objectives, capabilities, features, constraints, and top-level functions. The formal requirements are contained in the Specific Requirements sections of the Use Cases, the Supplementary Requirements Specification, and the Traditional Requirements Specification.

4.12.8 Hybrid Process Example

Here we give a simple example of a use case and a use case diagram (Figure 4.15) that helps illustrate the concepts presented in this section. This example is based on the Microwave Oven Software Product Line Case Study given in Gomaa (2004), which contains some very nice use case modeling examples. This one was selected due to its simplicity. Two other examples of implementing this hybrid process for capturing requirements are available at the Umpire's Assistant (2005) and the Spin Coach (2006).

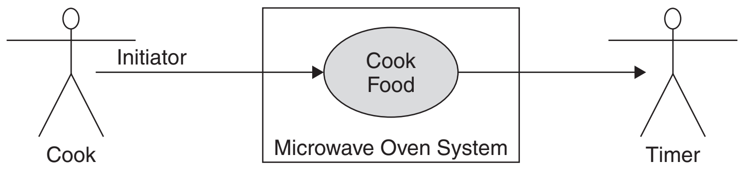

Figure 4.15 Use case diagram for a microwave oven system. (Copyright © 2004, A. T. Bahill, from http://www/sie.arizona.edu/sysengr/slides/. Used with permission.)

Use Case Name: Cook Food.

Brief Description: This use case describes the user interaction and operation of a microwave oven. Cook invokes this use case in order to cook food in the microwave.

Added Value: Food is cooked.

Scope: A microwave oven.

Primary Actor: Cook.

Supporting Actors: Timer.

Preconditions: The microwave is waiting to be used.

Main Success Scenario

1. Cook opens the microwave oven door, puts food into the oven, and then closes the door.
2. Cook specifies a cooking time.
3. System displays entered cooking time to Cook.
4. Cook starts the system.
5. System cooks the food and continuously displays the remaining cooking time.
6. Timer indicates that the specified cooking time has been reached and notifies the system.
7. System stops cooking and displays a visual and audio signal to indicate that cooking is completed.
8. Cook opens the door, removes food, and then closes the door.
9. System resets the display [exit use case].

Alternate Flows

4a. If Cook attempts to start the system without closing the door, then the system will not start.

4b. If Cook attempts to start the system without placing food inside, then the system will not start.

4c. If Cook enters a cooking time equal to zero, then the system will not start.

5a. If Cook opens the door during cooking, the system will stop cooking. Cook can either close the door and restart the system (continue at step 5), or Cook can cancel cooking.

5b. Cook cancels cooking. The system stops cooking. Cook may start the system and resume at step 5. Alternatively, Cook may reset the microwave to its initial state (cancel timer and clear displays).

Author: Hassan Gomaa.

Last Changed: October 23, 2004.

Specific Requirements

Functional Requirements

Req. F1: The system shall provide a mechanism for Cook to enter a cooking time [from step 2].

Req. F2: The system shall be capable of displaying the cooking time entered by Cook [from step 3].

Req. F3: The system shall cook food using microwave radiation [from step 5].

Req. F4: The system shall be capable of calculating and displaying the remaining cooking time [from step 5].

Req. F5: The system shall interface with a timer mechanism such that the system is stopped when the timer elapses [from step 6].

Req. F7: The system shall emit an audible signal when the timer has elapsed [from step 7].

Req. F8: The system shall indicate visually when the timer has elapsed [from step 7].

Req. F9: The system shall be capable of determining whether the oven door has been closed [from step 1a].

Req. F10: The system shall not start, if the system detects that the door is open [from step 4a].

Req. F11: The system shall be capable of determining if food has been placed in the oven [from step 4b].

Req. F12: The system shall not start, if the system detects that no food has been placed in the oven [from step 4b].

Req. F13: The system shall stop running if the oven door is opened while the system is running [from step 5a].

Req. F14: The system shall provide a mechanism to cancel a cooking time entered by Cook [from steps 5a and 5b].

Req. F15: The system shall stop running if the cooking time is canceled while the system is running [from steps 5a and 5b].

Nonfunctional Requirements

Req. N1: The system shall allow Cook to enter the cooking time in less than five keystrokes [refinement of Req. F1, obtained from stakeholder interviews].

Req. N2: The cooking time displayed by the system shall be visible to a person with 20/20 vision standing 5 feet from the oven in a room with a luminance level between 0 and 100 foot-candles [refinement of Req. F2, obtained from stakeholder interviews].

Req. N3: The system shall raise the temperature of food in the oven such that temperatures at two distinct locations in the food are different by less than 10% [refinement of Req. F3, obtained from stakeholder interviews where stakeholders desire even cooking of food].

Req. N4: The system shall update the remaining cooking time display every second [refinement of Req. F4].

Req. N5: The audible signal emitted by the oven shall have an intensity level of 80 decibels $\pm$ 2 decibels [refinement of Req. F7, obtained from stakeholder interviews].

Req. N6: The system shall detect food items weighing at least 0.05 ounce and with a volume of at least 1 cubic inch [refinement of Req. F11, obtained from stakeholder interviews].

Req. N7: The system shall comply with Section 1030 of Title 21—Food and Drugs, Chapter I—Food and Drug Administration, Department of Health and Human Services, Subchapter J: Radiological Health [refinement of Req. F3, specifies required compliance with health standards].

Req. N8: The system shall provide a minimum of $14\frac{3}{4}$-in width × $8\frac{3}{4}$-in. height × $15\frac{3}{4}$-in. depth inside cooking area volume [from step 1, obtained from stakeholder interviews].

Req. N9: The cooking time shall be adjustable between 1 second and 90 minutes [refinement of Req. F1, obtained from stakeholder interviews].

4.12.9 Hybrid Process Summary

Use case models and traditional shall-statement requirements are synergistic specification techniques that should be employed in a complementary fashion to best communicate and document requirements. Are use cases requirements? Not exactly—they are a vehicle to discover requirements. The requirements are actually embedded within the use case's textual description, making use cases a container for the requirements. How do use cases relate to a traditional requirements specification? Use cases provide context for requirements that are documented using shall-statement notation in a traditional requirements specification. These shall-statement requirements can be extracted from a use case's narrative. Where are the requirements actually documented? We suggest that the requirements be documented using the requirements specification structure illustrated in Figure 4.13. This structure includes (1) a Use Case Model for capturing requirements that are associated with individual use cases and (2) a Supplementary Requirements Specification for capturing system-wide requirements. The specific contribution of the Daniels and Bahill (2004) paper was the introduction of a Functional Requirements part to the Specific Requirements section. This Functional Requirements part contains the functional requirements written in formal shall-statement language. A problem with the hybrid process is that it produces specific, low-level, design requirements. We do not know if these can be abstracted to high-level customer requirements or goals. However, the brief description and added value slots of the use ease might suggest the high-level requirements and goals. This section has shown an example illustrating the hybrid process for combining use case models with traditional shall-statement requirements.

System design often starts with the use cases. Then, as shown in this section, the requirements can be elicited in the use cases. Sequence diagrams can be drawn to solidify the use case models. Then classes will be formed. The functional requirements help the designer to identify the functions or operations of each class. Finally, the relationships between classes are defined. This now produces a model on which the system can be designed.

In summary, the hybrid requirements process captures shall-statement requirements in the use case Sequences of Events. These functional requirements are put in the Specific Requirements section of the Use Case Report. These functional requirements can also be put into the class diagrams and a requirements database to support traditional specs and tracing.

4.13 CONCLUSION

Customer dissatisfaction, cost overruns, and schedule slippage are often caused by poor requirements that are produced by people who do not understand the requirements process. This chapter provides a high-level overview of the system requirements process and explains types, sources, and characteristics of good requirements. System requirements, however, are seldom stated by the customer. Therefore, this chapter shows ways to help you work with your customer to *discover* the system requirements. It explains terminology commonly used in the requirements development field, such as verification, validation, technical performance measures, and the various design reviews. It also presents the hybrid process for capturing requirements in use cases. Developing executable models is often a part of the requirements process.

ACKNOWLEDGMENTS

Rob Culver and Bill Wuersch at BAE Systems in San Diego contributed ideas about requirements. Matt Bishop at Boeing in Seattle made suggestions for this chapter. An earlier version of this chapter is available at http://www.sie.arizona.edu/sysengr/requirements.

REFERENCES

Abadi, C. D., and Bahill, A. T. (2003). The difficulty in distinguishing product from process. *Syst. Eng.* **6**(2):108–115.

Adolph, S., and Bramble, P. (2003). *Patterns for Effective Use Cases*. Reading, MA: Addison-Wesley.

Armour, F., and Miller, G. (2001). *Advanced Use Case Modeling*. Reading, MA: Addison-Wesley.

Bahill, A. T., and Botta, R. (2008). Fundamental principles of good system design. *Eng. Manage. J.* **20**(4):44–52.

Bahill, A. T., and Chapman, W. L. (1993). A tutorial on quality function deployment. *Eng. Manage. J.* **5**(3):24–35.

Bahill, A. T., and Daniels, J. (2003). Using object-oriented and UML tools for hardware design: a case study. *Syst. Eng.* **6**(1):28–48.

Bahill, A. T., and Henderson, S. J. (2005). Requirements development, verification and validation exhibited in famous failures. *Syst. Eng.* **8**(1):1–14.

Bahill, A. T., and Karnavas, W. J. (1992). *Bat Selector*. U.S. Patent 5,118,102.

Bahill, A. T., and Karnavas, W. J. (2000). Risk analysis of a pinewood derby: a case study. *Syst. Eng.* **3**(3):143–155.

Bahill, A. T., Botta, R., and Daniels, J. (2006). The Zachman framework populated with baseball models. *J. Enterprise Archit.* **2**(4):50–68.

Bahill, A. T., Szidarovszky, F., Botta, R., and Smith, E. D. (2008). Valid models require defined levels. *Int. J. Gen. Syst.* **37**(5):533–571.

Bicknell, K. D., and Bicknell, B. A. (1994). *The Road Map to Repeatable Success: Using QFD to Implement Change*. Boca Raton, FL: CRC Press.

Booch, G., Rumbaugh, J., and Jacobson, I. (2005). *The Unified Modeling Language User Guide*. Reading, MA: Addison-Wesley.

Botta, R., and Bahill, A. T. (2006). A prioritization process. In: *Proceedings of 16th Annual International Symposium of INCOSE*, Orlando, FL, July 9–13.

Botta, R., and Bahill, A. T. (2007). A prioritization process. *Eng. Manage. J.* **19**(4):20–27.

Botta, R., Bahill, Z., and Bahill, A. T. (2006). When are observable states necessary? *Syst. Eng.* **9**(3):228–240.

Chapman, W. L., Bahill, A. T., and Wymore, W. (1992). *Engineering Modeling and Design*. Boca Raton, FL: CRC Press.

Chapman, W. L., and Bahill, A. T. (1996). Design modeling and production. In: *The Engineering Handbook* (R. C. Dorf, ed.), pp. 1732–1737. Boca Raton, FL: CRC Press.

CMMI for Development, ver. 1.2 (2006). Software Engineering Institute, Pittsburgh, PA, http://www.sei.cmu.edu/cmmi/, retrieved December 2008.

Cockburn, A. (2001). *Writing Effective Use Cases*. Reading, MA: Addison-Wesley.

Daniels, J., and Bahill, A. T. (2004). Hybrid process combines traditional requirements and use cases. *Syst. Eng.* **7**(4):303–319.

Daniels, J., Werner, P. W., and Bahill, A. T. (2001). Quantitative methods for tradeoff analyses. *Syst. Eng.* **4**(3):190–212.

Davis, R., and Buchanan, B. G. (1984). Meta-level knowledge. In: *Rule-Based Expert Systems, The MYCIN Experiments of the Stanford Heuristic Programming Project* (B. G. Buchanan and X. Short liffe, eds.), pp. 507–530. Reading, MA: Addison-Wesley.

Deming, W. E. (1982). *Out of the Crisis*. Cambridge, MA: MIT Center for Advanced Engineering Study.

Feynman, R. (1949). Space–time approach to quantum electrodynamics. *Phys. Rev.*, **76**:769–789.

Gause, D. C., and Weinberg, G. M. (1990). *Are Your Lights On? How to Figure Out What the Problem Really Is*. New York: Dorset House Publishing.

Gomaa, H. (2004). *Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures*. Reading, MA: Addison-Wesley.

Grady, J. O. (1993). *System Requirements Analysis*. New York: McGraw-Hill.

Grady, J. O. (1994). *System Integration*. Boca Raton, FL: CRC Press.

Hooks, I. (1994). Writing good requirements. In: *Proceedings NCOSE*, pp. 197–203.

Hooks, I. F., and Farry, K. A. (2001). *Customer-Centered Products: Creating Successful Products Through Smart Requirements Management*. New York: AMACOM.

Jacobson, I., (2000). *Use Cases in Large-Scale Systems, Road to the Unified Process*. Cambridge, UK: Cambridge University Press.

Jacobson, I., Ericsson, M., and Jacobson, A. (1995). *The Object Advantage: Business Process Reengineering with Object Technology*. Reading, MA: Addison-Wesley.

Jacobson, I., Booch, G., and Rumbaugh, J. (1999). *The Unified Software Development Process*. Reading, MA: Addison-Wesley.

Kano, N. (1993). Special issue on Kano's methods for understanding customer-defined quality, *Center of Quality Management Journal* Vol. 2, No. 4, Fall 1993 (PDF - 350k).

Karnavas, W. J., Sanchez, P., and Bahill, A. T. (1993). Sensitivity analyses of continuous and discrete systems in the time and frequency domains. *IEEE Trans. Syst. Man Cybern.* **SMC-23**:488–501.

Kerzner, H. (1995). *Project Management: A Systems Approach to Planning, Scheduling, and Controlling*. New York: Van Nostrand Reinhold.

Kulak, D., and Guiney, E. (2000). *Use Cases: Requirements in Context*. Reading, MA: Addison-Wesley.

Leffingwell, D., and Widrig, D. (2000). *Managing Software Requirements*. Reading, MA: Addison-Wesley.

LaPlue, L., Garcia, R. A., and Rhodes, R. (1995). A rigorous method for formal requirements definition. In: *Systems Engineering in the Global Market Place*, Proceedings of the Fifth Annual Symposium of the National Council on Systems Engineering, July 22–26, St. Louis, pp. 401–406.

Latzko, W. J., and Saunders, D. M. (1995). *Four Days with Dr. Deming*. Reading, MA: Addison-Wesley.

Martin, J. (1996). *Systems Engineering Guideline*. Boca Raton, FL: CRC Press.

Moody, J. A., Chapman, W. L., Van Voorhees, F. D., and Bahill, A. T. (1997). *Metrics and Case Studies for Evaluating Engineering Designs*. Upper Saddle River, NJ: Prentice Hall PTR.

NASA. (2000). *Mars Program Independent Assessment Team Summary Report*. Washington, DC: NASA. March 14, 2000.

Oakes, J., Botta, R., and Bahill, A. T. (2006). Technical performance measures. In: *Proceedings of 16th Annual International Symposium of INCOSE*, Orlando, FL, July 9–13, 2006.

Object Management Group (OMG). (2007). *UML Resource Page*. Available at http://www.uml.org.

Övergaard, G., and Palmkvist, K. (2005). *Use Cases: Patterns and Blueprints*. Reading, MA: Addison-Wesley.

Rechtin, E., and Maier, M. (1996). *Systems Architecting.* Boca Raton, FL: CRC Press.

Sage, A. P. (1992). *Systems Engineering*. Hoboken, NJ: Wiley.

Shand, R. M. (1994). User manuals as project management tools. *IEEE Trans. Prof. Commun.* **37**:75–80, 123–142.

Shishko, R., and Chamberlain, R. G. (1995). *NASA Systems Engineering Handbook*, SP-6105.

Simon, H. A. (1957). *Models of Man: Social and Rational*. Hoboken, NJ: Wiley.

Smith, E. D., Szidarovszky, F., Karnavas, W. J., and Bahill, A. T. (2008). Sensitivity analysis, a powerful system validation technique, *The Open Cybernetics and Systemics Journal*, http://www.bentham.org/open/tocsj/openaccess2.htm, **2**:39–56, doi: 10.2174/1874110X00802010039.

Smith, E. D., and Bahill, A. T. (2007). Risk analysis. In: *Proceedings of 17th Annual International Symposium of INCOSE*, San Diego, CA, June 24–28.

Smith, E. D., Son, Y. J., Piattelli-Palmarini, M., and Bahill, A. T. (2007). Ameliorating mental mistakes in tradeoff studies. *Syst. Eng.* **10**(3):222–240.

Sommerville, I. (1989). *Software Engineering.* Reading, MA: Addison-Wesley.

Spin Coach (2006). Available at http://www.sie.arizona.edu/sysengr/sie554/SpinCoach.doc.

Umpire's Assistant (2005). Available at http://www.sie.arizona.edu/sysengr/sie577/UmpiresAssistant.doc.

Wymore, W. (1993). *Model-Based Systems Engineering*. Boca Raton, FL: CRC Press.

Young, R. R. (2001). *Effective Requirements Practices*. Reading, MA: Addison-Wesley.

Young, R. R. (2006). Criteria of a good requirement. Available at http://www.ralphyoung.net/artifacts/CriteriaGoodRequirement.pdf.