Chapter 1

Introduction to Requirements Engineering

Motivation

Very early in the drive to industrialize software development, Royce (1975) pointed out the following truths:

> There are four kinds of problems that arise when one fails to do adequate requirements analysis: top-down design is impossible; testing is impossible; the user is frozen out; management is not in control. Although these problems are lumped under various headings to simplify discussion, they are actually all variations of one theme—poor management. Good project management of software procurements is impossible without some form of explicit (validated) and governing requirements.

Even though top-down design has been largely displaced by object-oriented design, these truths still apply today.

A great deal of research has verified that devoting systematic effort to requirements engineering can greatly reduce the amount of rework needed later in the life of the software product and can improve various qualities of the software cost effectively. Too often systems engineers forgo sufficient requirements engineering activity either because they do not understand how to do requirements engineering properly or because there is a rush to start coding (in the case of a software product). Clearly, these eventualities are undesirable, and it is a goal of this book to help engineers understand the correct principles and practices of requirements engineering.

What Is Requirements Engineering?

There are many ways to portray the discipline of requirements engineering depending on the viewpoint of the definer. For example, a bridge is a complex system, but has a relatively small number of patterns of design that can be used (e.g., suspension, trussed, cable-stayed). Bridges also have specific conventions and applicable regulations in terms of load requirements, materials that are used, and the construction techniques employed. So when speaking with a customer (e.g., the state department of transportation) about the requirements for a bridge, much of its functionality can be captured succinctly:

> The bridge shall replace the existing span across the Brandywine River at Creek Road in Chadds Ford, Pennsylvania, and shall be a cantilever bridge of steel construction. It shall support two lanes of traffic in each direction and handle a minimum capacity of 100 vehicles per hour in each direction.

Of course there is a lot of information missing from this "specification," but it substantially describes what this bridge is to do and the design choices available to achieve these requirements are relatively simple.

Other kinds of systems, such as biomechanical or nanotechnology systems with highly specialized domain language, have seemingly exotic requirements and constraints. Still other complex systems have so many kinds of behaviors that need to be embodied (even word-processing software can support thousands of functions) that the specification of said systems becomes very challenging indeed.

Inasmuch as the author is a software engineer, we reach out to that discipline for a convenient, more-or-less universal definition for requirements engineering that is due to Pamela Zave (1997, p. 315):

> Requirements engineering is the branch of software engineering concerned with the *real-world goals* for, functions of, and constraints on software systems. It is also concerned with the relationship of these factors to *precise specifications* of software behavior, and to their *evolution over time and across software families.*

But we wish to generalize the notion of requirements engineering to include any system, whether it be software only, hardware only, or hardware and software (and many complex systems are a combination of hardware and software), so we rewrite Zave's definition as follows:

> Requirements engineering is the branch of **engineering** concerned with the *real-world goals* for, functions of, and constraints on, **systems**. It is also concerned with the relationship of these factors to *precise*

specifications of **system** behavior and to their *evolution over time and across families of* ***related systems***.

The changes we have made to Zave's definition are in **bold**. We refer to this modified definition when we speak of "requirements engineering" throughout this text. And we explore all the ramifications of this definition and the activities involved in great detail as we move forward.

You Probably Don't Do Enough Requirements Engineering

Research suggests that requirements engineering is not well done in industry. For example, in a 2003 study of nearly 2,000 software and systems practitioners in southeast Pennsylvania, 52% of respondents indicated that they believed that their company did not do enough requirements engineering (Neill and Laplante 2003).

In a 2008 follow-up survey of companies in the same region, the situation had not changed: 52% of respondents reported that their companies "did not perform an adequate amount of requirements engineering" (Marinelli 2008).

Finally, in an independent study of seven embedded systems engineering companies in Europe, Sikora, Tenbergen, and Pohl (2012) found similar results; in particular, they concluded that "existing requirements engineering methods are insufficient for handling requirements for complex embedded systems."

One potential reason for inadequate requirements engineering is suggested by Wnuk, Regnell, and Berenbach (2011) who observed that companies seem to have difficulty scaling up requirements engineering practices in terms of activity scope and the structure of requirements artifacts. In their survey, Sikora, Tenbergen, and Pohl (2012) found another possible reason for the apparent inadequacy in industrial requirements engineering. They found that practitioners desire "requirements specifications that are properly integrated into the overall systems architecture but are solution-free with regard to the specified function or component itself." Survey respondents noted that existing method support is inadequate in promoting this goal. It seems, then, that requirements engineering research has a long way to go in meeting the needs of practitioners.

What Are Requirements?

Part of the challenge in requirements engineering has to do with an understanding of what a "requirement" really is. Requirements can range from high-level, abstract statements and back-of-the-napkin sketches to formal (mathematically rigorous) specifications. These varying representation forms occur because stakeholders have needs at different levels, hence, depend on different abstraction representations.

Stakeholders also have varying abilities to make and read these representations (e.g., a business customer versus a design engineer), leading to diverse quality in the requirements. We discuss the nature of stakeholders and their needs and capabilities in the next chapter.

Requirements versus Goals

A fundamental challenge for the requirements engineer is recognizing that customers often confuse requirements and goals (and engineers sometimes do too).

Goals are high-level objectives of a business, organization, or system, but a *requirement* specifies how a goal should be accomplished by a proposed system. So, to the state department of transportation, a goal might be "to build the safest bridge in the world." What is really intended, however, is to stipulate performance requirements on the bridge materials, qualifications of the contractor and engineers, and building techniques that will lead to a safe bridge.

To treat a goal as a requirement is to invite trouble because achievement of the goal will be difficult to prove. In addition, goals evolve as stakeholders change their minds and refine and operationalize goals into behavioral requirements.

Requirements Level Classification

To deal with the diversity in requirements types, Sommerville (2005) suggests organizing them into three levels of abstraction:

- User requirements
- System requirements
- Design specifications

User requirements are abstract statements written in natural language with accompanying informal diagrams. They specify what services (user functionality) the system is expected to provide and any constraints. Collected user requirements often appear as a "concept of operations" (Conops) document. In many situations user stories can play the role of user requirements.

System requirements are detailed descriptions of the services and constraints. System requirements are sometimes referred to as the *functional specification* or *technical annex* (a term that is rarely used). These requirements are derived from analysis of the user requirements and they should be structured and precise. The requirements are collected in a systems requirements specification (SRS) document. Use cases can play the role of system requirement in many situations.

Finally, design specifications emerge from the analysis and design documentation used as the basis for implementation by developers. The system design specification is essentially derived directly from analysis of the system requirements

To illustrate the differences in these specification levels, consider the following from the airline baggage handling system:

A user requirement:

- The system shall be able to process 20 bags per minute.

Some related system requirements.

- Each bag processed shall trigger a baggage event.
- The system shall be able to handle 20 baggage events per minute.

Finally, the associated system specifications are

1.2 The system shall be able to process 20 baggage events per minute in operational mode.
  1.2.1 If more than 20 baggage events occur in a one-minute interval, then the system shall . . .
  1.2.2 [more exception handling] . . .

For the pet store POS system, consider the following:

A user requirement:

- The system shall accurately compute sale totals including discounts, taxes, refunds, and rebates; print an accurate receipt; and update inventory counts accordingly.

Some related system requirements:

- Each sale shall be assigned a sales ID.
- Each sale may have one or more sales items.
- Each sale may have one or more rebates.
- Each sale may have only one receipt printed.

Finally, the associated software specifications are:

1.2 The system shall assign a unique sales ID number to each sale transaction.
  1.2.1 Each sales ID may have zero or more sales items associated with it, but each sales item must be assigned to exactly one sales ID.

The systems specifications in the appendices also contains numerous specifications organized by level for your exploration.

The different specification levels guide progressive testing throughout the project life cycle (Figure 1.1). Thus, user requirements, which are discovered first, are

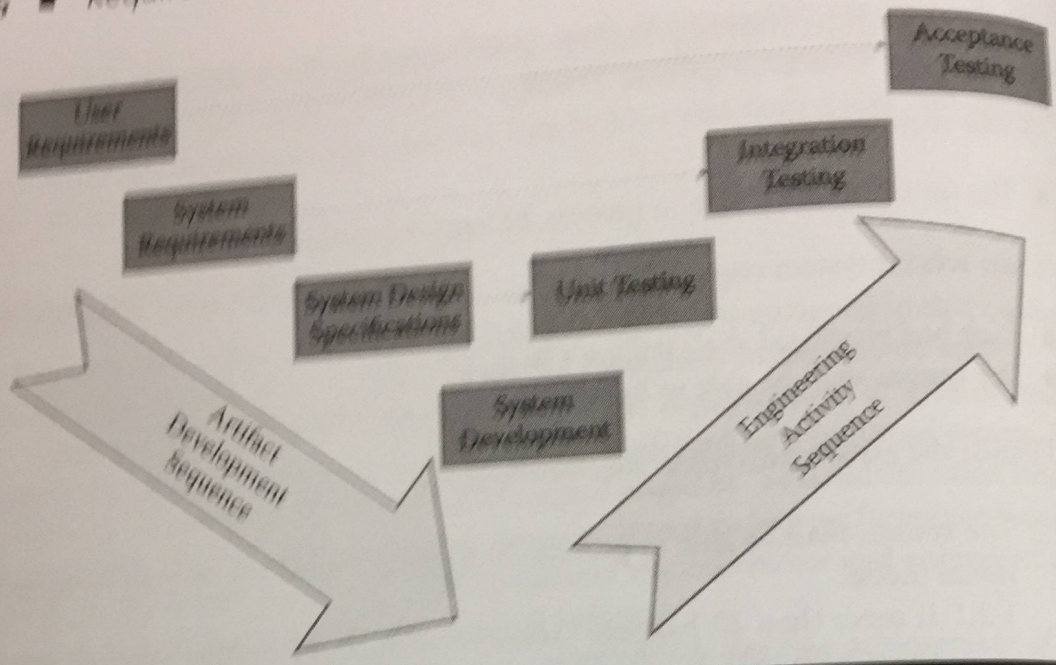

Figure 1.1 Relationship between requirements specification levels and testing.

used as the basis for final acceptance testing. The systems requirements, which are generally developed after the user requirements, are used as the basis for the integration testing that precedes acceptance testing. And finally, the design specifications that are derived from the system requirements, are used for unit testing after the system has been implemented.

Requirements Specifications Types

Another taxonomy for requirements specifications focuses on the type of requirement from the following list of possibilities:

- Functional requirements
- Nonfunctional requirements (NFRs)
- Domain requirements

Let's look at these more closely.

Functional Requirements

Functional requirements describe the services the system should provide and how the system will react to its inputs. In addition, the functional requirements need to state explicitly certain behaviors that the system should not do (more on this later). Functional requirements can be high level and general (in which case they are user requirements in the sense that was explained previously) or they can be detailed,

expressing inputs, outputs, exceptions, and so on (in which case they are the system requirements described before).

There are many forms of representation for functional requirements, from natural language (in our case, the English language), visual models, and the more rigorous formal methods. We spend much more time discussing requirements representation in Chapter 4.

To illustrate some functional requirements, consider the following set for the baggage handling system:

1.1 The system shall handle up to 20 bags per minute.
1.4 When the system is in idle mode, the conveyor belt shall not move.
1.8 If the main power fails, the system shall shut down in an orderly fashion within 5 seconds.
1.41 If the conveyor belt motor fails, the system shall shut down the input feed mechanism within 3 seconds.

For the pet store POS system, the following might be some functional requirements:

4.1 When the operator presses the "total" button, the current sale enters the closed-out state.
4.1.1 When a sale enters the closed-out state, a total for each non-sale item is computed as number of items times the list price of the item.
4.1.2 When a sale enters the closed-out state, a total for each sale item is computed.

Nonfunctional Requirements

Nonfunctional requirements (NFRs) are imposed by the environment in which the system is to operate. These kinds of environments include timing constraints, quality properties, standard adherence, programming languages to be used, and so on.

Sommerville (2005) suggests the nonfunctional types depicted in Figure 1.2, and we review a few of these. Many of these NFRs are beyond the control of the requirements engineer or customer. For example, in the United States, privacy requirements are typically based on certain legislation (such as the Health Information Portability and Accountability Act or HIPAA), as are safety requirements. Standards requirements are based on compliance with national or international standards. Interoperability requirements may be partially under the control of the requirements engineer and customer if the system in question is to operate in conjunction with other systems controlled by the customer (if a third party controls the other systems, then interoperability is an independent variable). Other nonfunctional requirements, such as delivery, usability, performance, and organization,

Figure 1.2 Nonfunctional requirement types (Redrawn from I. Sommerville. *Software Engineering, 7th edition. Boston, MA:* Addison-Wesley. 2005.)

may be under the
standards or the
functional require
using an appropri
To illustrate
system. Some N

Product requ

- Efficiency
  - Perfo
  - Space
- Reliabilit
- Portabili
- Usability
- Organiz
- Delivery
  updates
- Implem
- Standa
- Extern
- Interop
- Ethica
- Legisl
  - Pr
- Safety

For the
Produc

- Effic
  -
  -
- Rel
- Por
- Us
- Or
- De
- In

* Mean
† Mean
‡ Federa
§ Occup

may be under the control of the customer but might also be driven by industry standards or the need for competitive advantage. Whatever the case, all of the nonfunctional requirements need to be tracked by the requirements engineer, typically using an appropriate software tool.

To illustrate some nonfunctional requirements, consider the baggage handling system. Some NFRs might include:

Product requirements

- Efficiency
  - Performance (e.g., number of bags per minute)
  - Space (e.g., physical size of system, amount of memory, power consumption)
- Reliability (MTBF,* MTFF†)
- Portability (e.g., if it can be used with other hardware)
- Usability (amount of training required)
- Organizational requirements
- Delivery (e.g., date of delivery, date when fully operational, training sessions, updates)
- Implementation (e.g., full capability in first roll-out or phased capability)
- Standards (if there are industry standards for baggage handling systems)
- External requirements
- Interoperability (e.g., with other equipment, communications standards)
- Ethical (e.g., security clearance for REs, professional certification)
- Legislative
  - Privacy (e.g., HIPAA, FERPA‡)
- Safety (various OSHA§ regulations)

For the Pet Store POS system we might have these NFRs:
Product requirements

- Efficiency
  - Performance (e.g., time to finalize a sale transaction)
  - Space (e.g., physical size of system, amount of memory, power consumption)
- Reliability (MTBF, MTFF)
- Portability (e.g., if it can be used with other hardware)
- Usability (amount of training required)
- Organizational requirements
- Delivery (e.g., date of delivery, fully operational, training sessions, updates)
- Implementation (e.g., full capability in first roll-out or phased capability)

---

* Mean time before failure.
† Mean time before first failure.
‡ Federal Educational Rights and Privacy Act (US).
§ Occupational Safety and Health Administration (US).

- Standards (if there are industry standards for POS systems)
- External requirements
- Interoperability (e.g., with other equipment, communications standards)
- Ethical (e.g., professional certification for sales associates)
- Legislative
  - Privacy (e.g., general laws governing credit card transactions)
- Safety (various OSHA regulations)

To conclude this discussion, we note that some NFRs are difficult to define precisely, making them difficult to verify. Moreover, it is easy to confuse goals with NFRs.

Remember a goal is a general intention of a stakeholder, for example:

> The system should be easy to use by experienced operators,

whereas a verifiable NFR is a statement using some objective measure:

> Experienced operators shall be able to use the following 18 system features after two hours of hands-on instructor-led training with an error rate of no greater than 0.5%.

Here there is specificity as to which features an "experienced" user should master (although they are not listed here for brevity), a definition of the kind of training that is required (instructor led, as opposed to, say, self-study), and sufficient leeway to recognize that no human is perfect.

Domain Requirements

Domain requirements are derived from the application domain. These types of requirements may consist of new functional requirements or constraints on existing functional requirements, or they may specify how particular computations must be performed.

In the baggage handling system, for example, various domain realities create requirements. There are industry standards (we wouldn't want the new system to underperform versus other airlines' systems). There are constraints imposed by existing hardware available (e.g., conveyor systems). And there may be constraints on performance mandated by collective bargaining agreements with the baggage handlers union.

For the pet store POS system, domain requirements are imposed by conventional store practices. For example:

- Handling of cash, credit cards, and coupons
- Display interface and receipt format

- Conventions in the pet store industry (e.g., frequent-buyer incentives, buy one get one free)
- Sale of items by the pound (e.g., horse feed) versus by item count (e.g., dog leashes)

Appendices A and B also contain other examples of nonfunctional requirements.

Domain Vocabulary Understanding

The requirements engineer must be sure to fully understand the application domain vocabulary as there can be subtle and profound differences in the use of terms in different domains. The following true incident illustrates the point. The author was once asked to provide consulting services to a very large international package delivery company. After a few hours of communicating with the engineers of the package delivery company, it became clear that the author was using the term "truck" incorrectly. Whereas the author believed that the "truck" referred to the familiar vehicle that would deliver packages directly to customers, the company used the term "package car" to refer to those vehicles. The term "truck" was reserved to mean any long-haul vehicle (usually an 18-wheeled truck) that carried large amounts of packages from one distribution hub to another. So there was a huge difference in the volume of packages carried in a "truck" and a "package car." Imagine, then, if a requirement were written involving the processing of "packages from 1,000 trucks" (when what was really meant was "1,000 package cars"). Clearly this difference in domain terminology understanding would have been significant and potentially costly.

Requirements Engineering Activities

The requirements engineer is responsible for a number of activities. These include:

- Requirements elicitation/discovery
- Requirements analysis and reconciliation
- Requirements representation/modeling
- Requirements verification and validation
- Requirements management

We explore each of these activities briefly in the following sections and substantially more so in subsequent chapters.

Requirements Elicitation/Discovery

Requirements elicitation/discovery involves uncovering what the customer needs and wants. But elicitation is not like picking low-hanging fruit from a tree. Although some requirements will be obvious (e.g., the POS system will have to compute sales

tax), many requirements will need to be extricated from the customer through well-defined approaches. This aspect of requirements engineering also involves discovering who the stakeholders are; for example, are there any hidden stakeholders? Elicitation also involves determining the nonfunctional requirements, which are often overlooked.

Requirements Analysis and Agreements

Requirements analysis and requirements agreements involve techniques to deal with a number of problems with requirements in their "raw" form; that is, after they have been collected from the customers. Problems with raw requirements include:

- They don't always make sense.
- They often contradict one another (and not always obviously so).
- They may be inconsistent.
- They may be incomplete.
- They may be vague or just wrong.
- They may interact and be dependent on each other.

Many of the elicitation techniques that we discuss subsequently are intended to avoid or alleviate these problems. Formal methods are also useful in this regard. Requirements analysis and agreements are discussed in Chapter 4.

Requirements Representation

Requirements representation (or modeling) involves converting the requirements processed from raw requirements into some model (usual natural language, math, and visualizations). Proper representations facilitate communication of requirements and conversion into a system architecture and design. Various techniques are used for requirements representation including informal (e.g., natural language, sketches, and diagrams), formal (mathematically sound representations), and semi-formal (convertible to a sound representation or is partially rigorous). Usually some combination of these is employed in requirements representation, and we discuss these further in Chapters 4 and 6.

Requirements Validation

Requirements validation is the process of determining if the specification is a correct representation of the customers' needs. Validation answers the question, "Am I building the right product?" Requirements validation involves various semi-formal and formal methods, text-based tools, visualizations, inspections, and so on and is discussed in Chapter 5.

Requirements Management

One of the most overlooked aspects of requirements engineering, requirements management involves managing the realities of changing requirements over time. It also involves fostering traceability through appropriate aggregation and subordination of requirements and communicating changes in requirements to those who need to know.

Managers also need to learn the skills to "push back" intelligently when scope creep ensues. Using tools to track changes and maintain traceability can significantly ease the burden of requirements management. We discuss software tools to aid in requirements engineering in Chapter 8 and requirements management overall in Chapter 9.

Bodies of Knowledge

Three important structured taxonomies or bodies of knowledge exist for requirements engineering of software systems: the *Software Engineering Body of Knowledge* (SWEBOK 2012), the *Graduate Software Engineering Reference Curriculum* (GSwE 2009), and the *Principles and Practices (P&P) of Software Engineering Exam Specification* (Laplante, Kalinowski, and Thornton 2013). These bodies of knowledge focus on the discipline of software engineering, but they can be applied to the engineering of any kind of system, whether software, electrical, mechanical, or hybrid.

SWEBOK was established to promote a consistent view of software engineering and to provide a foundation for curriculum development and for certification and licensing examinations. It identifies the fundamental skills and knowledge that all graduates of a master's program in software engineering must possess.

The GSwE report highlights the strong link between software engineering and systems engineering and suggests that system engineering knowledge areas should be integrated into the software engineering curriculum (Kilcay-Ergin and Laplante 2012).

The Software Engineering P&P exam is used by certain states and jurisdictions of the United States as one component of licensure for those individuals who are working on software systems that affect the health, safety, and welfare of the public (Laplante, Kalinowski, and Thornton 2013). Inasmuch as the intent of the taxonomies are different, the scope of requisite knowledge also differs in some ways; for example, there is no clear coverage of GSwE's Initiation and Scope Definition in SWEBOK. A comparison of the scope of knowledge for these bodies of knowledge is shown in Table 1.1.

This text is intended to provide reasonable coverage of most the topics from the SWEBOK, however, it also substantially covers the other two bodies of knowledge. There are also emerging reference curricula and bodies of knowledge related to systems engineering, for example, the *Guide to the Systems Engineering Body of Knowledge* (SEBoK 2012) and the *Graduate Reference Curriculum for Systems Engineering* (GRCSE 2012). These also focus significantly on requirements engineering processes and activities.

Table 1.1 Comparison of GSwE, SEBOK, and Software Engineering P&P Knowledge Areas

GSwE 2009	*SWEBOK*	*Software Engineering P&P*
Fundamentals of RE • Relationship between systems engineering and software engineering • Definition of requirements • System design constraints • System design and requirements allocation • Product and process requirements • Functional and nonfunctional requirements • Emergent properties • Quantifiable requirements	Software Requirements Fundamentals • Definition of a software requirement • Product and process requirements • Functional and nonfunctional requirements • Emergent properties • Quantifiable requirements • System requirements and software requirements	Software Requirements Fundamentals • Concept of operations • Types of requirements • Product and process requirements • Functional and nonfunctional requirements • Quantifiable requirements • System requirements • Software requirements • Derived requirements • Constraints, service level
RE Process • Process models • Process actors • Process support and management • Process quality and improvement	Requirements Process • Process models • Process actors • Process support and management • Process quality and Improvement	
Initiation and Scope Definition • Determination and negotiation of requirements • Feasibility analysis • Process for requirements review/revision	No equivalent	
Requirements Elicitation • Requirements sources	Requirements Elicitation • Requirements sources • Elicitation techniques	Requirements Elicitation • Requirements sources • Elicitation techniques • Requirements

<table>
<tr>
<td>Requirements Elicitation
• Requirements sources
• Elicitation techniques</td>
<td>Requirements Elicitation
• Requirements sources
• Elicitation techniques</td>
<td rowspan="2">Requirements Elicitation
• Requirements sources
• Elicitation techniques
• Requirements representation</td>
</tr>
<tr>
<td>Requirements Analysis
• Requirements classification
• Conceptual modeling
• Heuristic methods
• Formal methods
• Requirements negotiation</td>
<td>Requirements Analysis
• Requirements classification
• Conceptual modeling
• Architectural design and requirements allocation
• Requirements negotiation</td>
</tr>
<tr>
<td>Requirements Specification
• Requirements specification techniques</td>
<td>Requirements Specification
• System definition document
• System requirements specification
• Software requirements specification</td>
<td>Requirements Specification
• System definition document
• System/subsystems specification
• Software requirements specification
• Interface requirements specification</td>
</tr>
</table>

(*Continued*)

Table 1.1 (*Continued*) Comparison of GSwE, SEBOK, and Software Engineering P&P Knowledge Areas

GSwE 2009	*SWEBOK*	*Software Engineering P&P*
Requirements Validation • Requirements reviews • Prototyping • Model validation • Acceptance tests	Requirements Validation • Requirements reviews • Prototyping • Model validation • Acceptance tests	Requirements Verification and Validation • Requirements reviews • Prototyping • Model validation • Simulation
Practical Considerations • Iterative nature of requirements process • Change management • Requirements attributes • Requirements tracing • Measuring requirements	Practical Considerations • Iterative nature of the requirements process • Change management • Requirements attributes • Requirements tracing • Measuring requirements	Requirements Management • Iterative nature of the requirements process • Change management • Requirement attributes • Requirements traceability • Measuring requirements • *Software requirements tools*

Source: Adapted from Ergin and Laplante (2012).

Requirements Er

What skills should a re
suggest that the requi
throughout the (softwa
to be general and whe
when necessary. Chris
engineer should be a g
a good negotiator, an
hard sciences and eng
skills). Finally, the req

Gorla and Lam (2
and "judging" in the
mean that requireme
information gathered
than leaving things c

Finally, Ebert (2
tency in the followir

- Requirements
- Systems engir
- Management
- Communicat
- Cognition
- Social interac

Ebert further n
petencies and that t
courses can, howe

Requirem

Another way to
various models
following role m

- Requirem
- Requirem
- Requirem
- Requirem
- The ignor

There are h

Requirements Engineer

What skills should a requirements engineer have? Christensen and Chang (1996) suggest that the requirements engineer should be organized, have experience throughout the (software) engineering life cycle, have the maturity to know when to be general and when to be specific, and be able to stand up to the customer when necessary. Christensen and Chang further suggest that the requirements engineer should be a good manager (to manage the process), a good listener, fair, a good negotiator, and multidisciplinary (e.g., have a background in traditional hard sciences and engineering augmented with communications and management skills). Finally, the requirements engineer should understand the problem domain.

Gorla and Lam (2004) imply that engineers should be "thinking," "sensing," and "judging" in the Myers–Briggs sense. We could interpret this observation to mean that requirements engineers are structured and logical (thinking), focus on information gathered and do not try to interpret it (sensing), and seek closure rather than leaving things open (judging).

Finally, Ebert (2010) suggests that requirement engineers should have competency in the following areas:

- Requirements engineering
- Systems engineering
- Management
- Communication
- Cognition
- Social interaction

Ebert further notes that academic programs are insufficient to develop these competencies and that they must be acquired through years of practice and study. Academic courses can, however, start new requirements engineers off in the right direction.

Requirements Engineering Paradigms

Another way to understand the nature of requirements engineering is to look at various models for the role of the requirements engineer. We have identified the following role models:

- Requirements engineer as software systems engineer
- Requirements engineer as subject matter expert
- Requirements engineer as architect
- Requirements engineer as business process expert
- The ignorant requirements engineer

There are hybrid roles for the requirements engineer based on the above as well.

Requirements Engineer as Software Systems Engineer

It is likely that many requirements engineers are probably former software systems designers or developers. When this is the case, the requirements engineer can positively influence downstream development of models (e.g., the software design). The danger in this case is that the requirements engineer may begin to create a design when she should be developing requirements specifications.

Requirements Engineer as Subject Matter Expert

In many cases the customer is looking to the requirements engineer to be a subject matter expert (SME) for expertise either in helping to understand the problem domain or in understanding the customers' own wants and desires. Sometimes the requirements engineer isn't an SME; he is an expert in requirements engineering. In those cases where the requirements engineer is not an SME, consider joining forces with an SME.

Requirements Engineer as Architect

Building construction is often used as a metaphor for software construction. In our experience architects and landscape architects play similar roles to requirements engineers (and this similarity is often used as an argument that software engineers need to be licensed). Daniel Berry has written about this topic extensively (Berry 1999, 2003). Berry noted that the analogous activities reduce scope creep and better involve customers in the requirements engineering process. In addition, Zachman (1987) introduced an architectural metaphor for information systems, although his model is substantially different from the one presented here.

The similarities between architecture (in the form of home specification) and software/systems specification are summarized in Table 1.2. If you have been through the process of constructing or renovating a home, you will appreciate the similarities.

Requirements Engineer as Business Process Expert

The activities of requirements engineering comprise a form of problem solving: the customer has a problem and the system must solve it. Often, solving the problem at hand involves the requirements engineer advising changes in business processes to simplify expression of system behavior. Although it is not the requirements engineer's role to conduct business process improvement, this side benefit is frequently realized.

Ignorance as Virtue

Berry (1995) suggested having both novices and experts in the problem domain involved in the requirements engineering process. His rationale is as follows. The "ignorant" people ask the "dumb" questions, and the experts answer these questions. Having the

Table 1.2 Arch

Home Building
Architect meet clients. Tours and pictures.
Architect mak (shows to clie
Architect ma elevations) a sophisticate cardboard, fly-through
Architect p additional
Future mo drawings)

requiremer
bad thing,
the hard
requireme

Berry
form of i
tion dom

Custo

What
custom

■
■
■
■

Table 1.2 Architectural Model for Systems Engineering

Home Building	*Software/System Building*
Architect meets with and interviews clients. Tours property. Takes notes and pictures.	Requirements engineer meets with customers and uses interviews and other elicitation techniques.
Architect makes rough sketches (shows to clients, receives feedback).	Requirements engineer makes models of requirements to show to customers (e.g., prototypes, draft SRS).
Architect makes more sketches (e.g., elevations) and perhaps more sophisticated models (e.g., cardboard, 3–D virtual models, fly-through animations).	Requirements engineer refines requirements and adds formal and semi-formal elements (e.g., UML). More prototyping is used.
Architect prepares models with additional detail (floor plans).	Requirements engineer uses information determined above to develop complete SRS
Future models (e.g., construction drawings) are for contractors' use.	Future models (e.g., software design documents) are for developers' use.

requirements engineer as the most "ignorant" person of the group is not necessarily a bad thing, at least with respect to the problem domain because it forces him to ask the hard questions and to challenge conventional beliefs. Of course, the "ignorant" requirements engineer is completely in opposition to the role of subject matter expert.

Berry also noted that using formal methods in requirements engineering is a form of ignorance because a mathematician is generally ignorant about an application domain before she starts modeling it.

Customer's Role

What role should the requirements engineer expect the customer to play? The customers' roles are many and include:

- Helping the requirements engineer understand what they need and want (elicitation and validation)
- Helping the requirements engineer understand what they don't want (elicitation and validation)
- Providing domain knowledge when necessary and possible
- Alerting the requirements engineer quickly and clearly when they discover that they or others have made mistakes

- Alerting the requirements engineer quickly when they determine that changes are necessary (really necessary)
- Controlling their urges to have "aha moments" that cause major scope creep
- Sticking to all agreements

In particular, the customer is responsible for answering the following four questions, with the requirements engineer's help, of course:

1. Is the system that I want feasible?
2. If so, how much will it cost?
3. How long will it take to build?
4. What is the plan for building and delivering the system?

The requirements engineer must manage customers' expectations with respect to these issues. We explore the nature and role of customers and stakeholders in the next chapter.

Problems with Traditional Requirements Engineering

Traditional requirements engineering approaches suffer from a number of problems, many of which we have already addressed (and others, which are addressed in the following) and many of these are not easily resolved. These problems include:

- Natural language problems (e.g., ambiguity, imprecision)
- Domain understanding
- Dealing with complexity (especially temporal behavior)
- Difficulties in enveloping system behavior
- Incompleteness (missing functionality)
- Overcompleteness (goldplating)
- Overextension (dangerous "all")
- Inconsistency
- Incorrectness
- And more

We explore the resolution of these problems throughout the text and particularly in Chapter 5.

Natural language problems result from the ambiguities and context sensitivity of natural (human) languages. We know these language problems exist for everyone, not just requirements engineers, and lawyers and legislators make their living finding, exploiting, or closing the loopholes found in any laws and contracts written in natural language.

Requirements Engineer

What skills should a requirements engineer have? Christensen and Chang (1996) suggest that the requirements engineer should be organized, have experience throughout the (software) engineering life cycle, have the maturity to know when to be general and when to be specific, and be able to stand up to the customer when necessary. Christensen and Chang further suggest that the requirements engineer should be a good manager (to manage the process), a good listener, fair, a good negotiator, and multidisciplinary (e.g., have a background in traditional hard sciences and engineering augmented with communications and management skills). Finally, the requirements engineer should understand the problem domain.

Gorla and Lam (2004) imply that engineers should be "thinking," "sensing," and "judging" in the Myers–Briggs sense. We could interpret this observation to mean that requirements engineers are structured and logical (thinking), focus on information gathered and do not try to interpret it (sensing), and seek closure rather than leaving things open (judging).

Finally, Ebert (2010) suggests that requirement engineers should have competency in the following areas:

- Requirements engineering
- Systems engineering
- Management
- Communication
- Cognition
- Social interaction

Ebert further notes that academic programs are insufficient to develop these competencies and that they must be acquired through years of practice and study. Academic courses can, however, start new requirements engineers off in the right direction.

Requirements Engineering Paradigms

Another way to understand the nature of requirements engineering is to look at various models for the role of the requirements engineer. We have identified the following role models:

- Requirements engineer as software systems engineer
- Requirements engineer as subject matter expert
- Requirements engineer as architect
- Requirements engineer as business process expert
- The ignorant requirements engineer

There are hybrid roles for the requirements engineer based on the above as well.

Requirements Engineer as Software Systems Engineer

It is likely that many requirements engineers are probably former software systems designers or developers. When this is the case, the requirements engineer can positively influence downstream development of models (e.g., the software design). The danger in this case is that the requirements engineer may begin to create a design when she should be developing requirements specifications.

Requirements Engineer as Subject Matter Expert

In many cases the customer is looking to the requirements engineer to be a subject matter expert (SME) for expertise either in helping to understand the problem domain or in understanding the customers' own wants and desires. Sometimes the requirements engineer isn't an SME; he is an expert in requirements engineering. In those cases where the requirements engineer is not an SME, consider joining forces with an SME.

Requirements Engineer as Architect

Building construction is often used as a metaphor for software construction. In our experience architects and landscape architects play similar roles to requirements engineers (and this similarity is often used as an argument that software engineers need to be licensed). Daniel Berry has written about this topic extensively (Berry 1999, 2003). Berry noted that the analogous activities reduce scope creep and better involve customers in the requirements engineering process. In addition, Zachman (1987) introduced an architectural metaphor for information systems, although his model is substantially different from the one presented here.

The similarities between architecture (in the form of home specification) and software/systems specification are summarized in Table 1.2. If you have been through the process of constructing or renovating a home, you will appreciate the similarities.

Requirements Engineer as Business Process Expert

The activities of requirements engineering comprise a form of problem solving: the customer has a problem and the system must solve it. Often, solving the problem at hand involves the requirements engineer advising changes in business processes to simplify expression of system behavior. Although it is not the requirements engineer's role to conduct business process improvement, this side benefit is frequently realized.

Ignorance as Virtue

Berry (1995) suggested having both novices and experts in the problem domain involved in the requirements engineering process. His rationale is as follows. The "ignorant" people ask the "dumb" questions, and the experts answer these questions. Having the

Table 1.2 Arch

Home Building
Architect meet clients. Tours and pictures.
Architect mak (shows to clie
Architect ma elevations) a sophisticate cardboard, fly-through
Architect p additional
Future mo drawings)

requiremer
bad thing,
the hard
requireme

Berry
form of i
tion dom

Custo

What
custom

■
■
■
■

Table 1.2 Architectural Model for Systems Engineering

Home Building	*Software/System Building*
Architect meets with and interviews clients. Tours property. Takes notes and pictures.	Requirements engineer meets with customers and uses interviews and other elicitation techniques.
Architect makes rough sketches (shows to clients, receives feedback).	Requirements engineer makes models of requirements to show to customers (e.g., prototypes, draft SRS).
Architect makes more sketches (e.g., elevations) and perhaps more sophisticated models (e.g., cardboard, 3–D virtual models, fly-through animations).	Requirements engineer refines requirements and adds formal and semi-formal elements (e.g., UML). More prototyping is used.
Architect prepares models with additional detail (floor plans).	Requirements engineer uses information determined above to develop complete SRS
Future models (e.g., construction drawings) are for contractors' use.	Future models (e.g., software design documents) are for developers' use.

requirements engineer as the most "ignorant" person of the group is not necessarily a bad thing, at least with respect to the problem domain because it forces him to ask the hard questions and to challenge conventional beliefs. Of course, the "ignorant" requirements engineer is completely in opposition to the role of subject matter expert.

Berry also noted that using formal methods in requirements engineering is a form of ignorance because a mathematician is generally ignorant about an application domain before she starts modeling it.

Customer's Role

What role should the requirements engineer expect the customer to play? The customers' roles are many and include:

- Helping the requirements engineer understand what they need and want (elicitation and validation)
- Helping the requirements engineer understand what they don't want (elicitation and validation)
- Providing domain knowledge when necessary and possible
- Alerting the requirements engineer quickly and clearly when they discover that they or others have made mistakes

- Alerting the requirements engineer quickly when they determine that changes are necessary (really necessary)
- Controlling their urges to have "aha moments" that cause major scope creep
- Sticking to all agreements

In particular, the customer is responsible for answering the following four questions, with the requirements engineer's help, of course:

1. Is the system that I want feasible?
2. If so, how much will it cost?
3. How long will it take to build?
4. What is the plan for building and delivering the system?

The requirements engineer must manage customers' expectations with respect to these issues. We explore the nature and role of customers and stakeholders in the next chapter.

Problems with Traditional Requirements Engineering

Traditional requirements engineering approaches suffer from a number of problems, many of which we have already addressed (and others, which are addressed in the following) and many of these are not easily resolved. These problems include:

- Natural language problems (e.g., ambiguity, imprecision)
- Domain understanding
- Dealing with complexity (especially temporal behavior)
- Difficulties in enveloping system behavior
- Incompleteness (missing functionality)
- Overcompleteness (goldplating)
- Overextension (dangerous "all")
- Inconsistency
- Incorrectness
- And more

We explore the resolution of these problems throughout the text and particularly in Chapter 5.

Natural language problems result from the ambiguities and context sensitivity of natural (human) languages. We know these language problems exist for everyone, not just requirements engineers, and lawyers and legislators make their living finding, exploiting, or closing the loopholes found in any laws and contracts written in natural language.

We have already covered the issue of domain understanding and have observed, as have others, that the requirements engineer may be an expert in the domain in which the system to be built will operate. System complexity is a pervasive problem that faces all systems engineers, and this is discussed shortly. The problems of fully specifying system behavior and of missing behavior are also very challenging, but there are some techniques that can be helpful in missing, at least, the most obvious functionality in a system.

Complexity

One of the greatest difficulties in dealing with the requirements engineering activities, particularly elicitation and representation, for most systems is that they are "complex." Without trying to invent a definition for "complexity," we contend that the challenges and complexity of capturing nontrivial behavior of any kind illustrate the notion of complex. Such difficulties are found in even the simplest of "repeatable" human endeavors.

Imagine, for example, someone asked you to describe your first five minutes of every morning from the moment you wake up. Could you do it with precision? (Try it.) No, you could not. Why? There are too many possible different paths your activities can take and too much uncertainty. Even with an atomic clock, you can't claim to wake up the same time every day (because even atomic clocks are imperfect). But of course, you wake up differently depending on the day of the week, whether you are on vacation from work, if it is a holiday, and so on. You have to account for that in your description. But what if you wake up sick? How does that change the sequence of events? What if you accidentally knock over the glass of water on your nightstand as you get up? Does that change the specification of the activity? Or if you trip on your dog as you head to the bathroom? We could go on with this example and repeat this exercise with other simple tasks such as mowing the lawn or shopping for food. In fact, until you constrain the problem to the verge of satire, you will find it challenging or even impossible to capture any nontrivial human activity precisely.

Now consider a complex information or embedded processing system. Such a system will likely need to have interactions with humans. Even in those systems that do not directly depend on human interaction, it is the intricacy of temporal behavior as well as the problems of unanticipated events that make requirements elicitation and specification so difficult (and all the other requirements activities hard too).

Rittel and Webber (1973) defined a class of complex problems that they called "wicked." Wicked problems have 10 characteristics:

- There is no definitive formulation of a wicked problem.
- Wicked problems have no stopping rule.
- Solutions to wicked problems are not true-or-false but good-or-bad.
- There is no immediate and no ultimate test of a solution to a wicked problem.

- Every solution to a wicked problem is a "one-shot operation"; because there is no opportunity to learn by trial and error, every attempt counts significantly.
- Wicked problems do not have an enumerable (or an exhaustively describable) set of potential solutions, nor is there a well-described set of permissible operations that may be incorporated into the plan.
- Every wicked problem is essentially unique.
- Every wicked problem can be considered a symptom of another problem.
- The existence of a discrepancy representing a wicked problem can be explained in numerous ways. The choice of explanation determines the nature of the problem's resolution.
- The planner (designer) has no right to be wrong.

Rittel and Webber (1973) meant wicked problems to be of an economic, political, and societal nature (e.g., hunger, drug abuse, conflict in the Middle East); therefore, they offer no appropriate solution strategy for requirements engineering. Nevertheless, it is helpful to view requirements engineering as a wicked problem because it helps explain why the task is so difficult: because in many cases, real systems embody most if not all of the characteristics of a wicked problem.

Four Dark Corners

Many of the problems with traditional requirements engineering arise from "four dark corners" (Zave and Jackson 1997, p. 2). We repeat the salient points here, verbatim, with commentary in italics.

1. All the terminology used in requirements engineering should be grounded in the reality of the environment for which a machine is to be built.
2. It is not necessary or desirable to describe (however abstractly) the machine to be built.
   Rather, the environment is described in two ways as it would be without or in spite of the machine and as we hope it will become because of the machine.
   *Specifications are the **what** to be achieved by the system, not the **how**.*
3. Assuming that formal descriptions focus on actions, it is essential to identify which actions are controlled by the environment, which actions are controlled by the machine, and which actions of the environment are shared with the machine.
   All types of actions are relevant to requirements engineering, and they might need to be described or constrained formally.

If formal descriptions focus on states, then the same basic principles apply in a slightly different form.
The method of formal representation should follow the underlying organization of the system. For example, a state-based system is best represented by a state-based formalization.

4. The primary role of domain knowledge in requirements engineering is in supporting refinement of requirements to implementable specifications.
Correct specifications, in conjunction with appropriate domain knowledge, imply the satisfaction of the requirements.
Failure to recognize the role of domain knowledge can lead to unfilled requirements and forbidden behavior.

Difficulties in Enveloping System Behavior

Imagine the arbitrary system shown in Figure 1.3 with n inputs and m outputs. Imagine that this system operates in an environment where the set of inputs, I, derives from human operators, sensors, storage devices, other systems, and so on. The outputs, O, pertain to display devices, actuators, storage devices, other systems, and so on. The only constraint we place on the system is that the inputs and outputs are digitally represented within the system (if they are from/to analog devices or systems, an appropriate conversion is needed). We define a behavior of the system as an input/output pair. Because the inputs and outputs are discrete, this system can be thought of as having an infinite but countable number of behaviors, $B \subseteq I \times O$.

Imagine the behavior space, B, is represented by the Venn diagram of Figure 1.3. The leftmost circle in the diagram represents the desired behavior set, as it is understood by the customer. The area outside this circle is unwanted behavior and desirable behavior that, for whatever reason, the customer has not discovered.

The requirements engineer goes about her business and produces a specification that is intended to be representative of the behavior desired by the customer (the rightmost circle of Figure 1.3). Being imperfect, this specification captures some (but not all) of the desired behavior, and it also captures some undesirable behavior. The desired behavior not captured in the specified behavior is the missing functionality. The undesirable behavior that is captured is forbidden functionality.

The goal of the requirements engineer, in summary, is to make the left and right circles overlap as much as possible and to discover those missing desirable behaviors that are not initially encompassed by the specification (leftmost circle).

Recognizing that there are an infinite number of behaviors that could be excluded from the desired behaviors of the system, we need to, nevertheless, focus

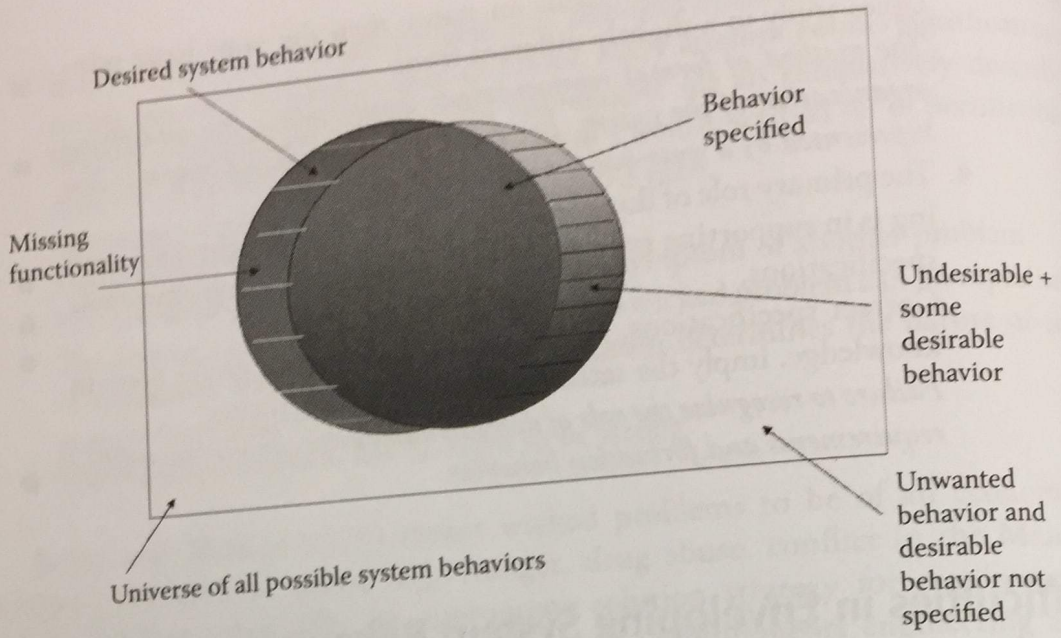

Figure 1.3 Universe of all possible behaviors, desired behavior, and specified behavior.

on the very worst of these. Therefore, some of the requirements that we will be discovering and communicating will take the form:

> The "system shall not . . ."

This unwanted functionality is sometimes called a "hazard," or "shall not behavior," or "forbidden behavior." The term "hazard" usually refers to forbidden behavior that can lead to substantial or catastrophic failure, whereas "forbidden behavior" has a less ominous connotation.

Known "shall nots" are not a problem, but what about unknown forbidden behavior? These unknown forbidden behaviors are related to unknown risks or uncertainties in the same way that the known forbidden behaviors are related to known risks (Voas and Laplante 2010). The following tragic vignette acts as an illustration of the difference between these two concepts.

On a snowy January 13th in 1982 an Air Florida Boeing 737 jet departed from Washington DC's National Airport and crashed into the 14th Street Bridge less than a mile from the airport. Of the 79 people onboard the plane, only five survived. The plane also struck seven vehicles on the bridge, killing four motorists and injuring another four.

The unfortunate passengers on the plane should have known that there was a risk of the plane crashing, no matter how small that risk may have been. In some

cases, the passengers may have even taken out special crash insurance. But what were the expectations of the motorists in the vehicles on the bridge who were killed or injured? Surely, the thought of an airplane crashing into the bridge never crossed their minds. For the motorists, therefore, this crash represented an unknown but manifested uncertainty.

As an example of a forbidden behavior, consider the following for the baggage handling system:

> When the "conveyor jam" signal is set to the high state, the feed mechanism shall not permit additional items to enter the conveyor system.

Similar forbidden behaviors can be generated for the POS system. We discuss how to identify and characterize these unwanted or "shall not" behaviors in Chapter 3.

Danger of "All" in Specifications

Requirement specification sentences often involve some universal quantification (e.g., "all users shall be able to access . . ."). But the use of "all" specifications is dangerous because they are usually not true (Berry and Kamsties 2000). For example, if a requirement for the baggage handling system states "All baggage shall be assigned a unique identifier," it is worthwhile to challenge this requirement. That is, is there some form of "baggage" that we wish not to give a unique identifier? If a suspicious package is given to an airport check-in agent, should they print out a barcode tag and put it on the package? Or, should they put it aside for further processing, such as inspection by security personnel? Whether the answer is "yes" or "no" is irrelevant; we want to consider these kinds of situations and deal with them as we are writing requirements in draft form rather than postpone the issue until requirements analysis occurs. Therefore requirements involving the word "all" should be challenged and relaxed when possible.

Related to "all" specifications are "never," "always," "none," and "each" (because they can be formally equated to universal quantification); these words and their mathematical equivalents should be avoided as well.

Exercises

1.1 What are some of the major objections and deterrents to proper requirements engineering activities?

1.2 How is requirements engineering different for "small" systems?

1.3 What are some factors that may cause customers to alter requirements?
1.4 What issues might arise when a requirements engineer, who is not a subject matter expert, enlists a subject matter expert to assist in defining requirements?
1.5 List some representative user requirements, system requirements, and software specifications for the pet store POS system.
1.6 List five typical functional requirements for the baggage handling system.
1.7 List five forbidden functional requirements for the pet store POS system.
1.8 Conduct some web research to discover prevailing regulations or standards (NFR) for smart home systems.
1.9 For the smart home system make a list of some of the hazards (what this system shall not do) based on the regulations you discovered in Exercise 1.8 and any other information you might have.
*1.10 A product line is a set of related products that share certain features. Product lines may be planned or they may emerge. What are some of the requirements engineering challenges for product lines? How does the "one-shot solution" aspect of Ritter and Webber's wicked problem apply? You may wish to conduct some research to answer this question.

References

Berry, D.M. (1995). The importance of ignorance in requirements engineering, *Journal of Systems and Software*, 28: 1, 179–184.

Berry, D.M. (1999). Software and house requirements engineering: Lessons learned in combating requirements creep, *Requirements Engineering Journal*, 3(3&4): 242–244.

Berry, D.M. (2003). More requirements engineering adventures with building contractors, *Requirements Engineering Journal*, 8(2): 142–146.

Berry, D.M. and Kamsties, E. (2000). The dangerous "all" in specifications. In *Proceedings of the Tenth International Workshop on Software Specification and Design (IWSSD'00)*, San Diego, CA, November 5–7.

Christensen, M. and Chang, C. (1996). Blueprint for the ideal requirements engineer, *Software*, March: 12.

Ebert, C. (2010). Requirements engineering: Management. In *Encyclopedia of Software Engineering*. P. Laplante (Ed.). Boca Raton, FL: Taylor & Francis, Published online: 932–948.

Gorla, N. and Lam, Y.W. (2004). Who should work with whom?: Building effective software project teams, *Communications of the ACM*, 47(6): 79–82.

Graduate Reference Curriculum for Systems Engineering (GRCSE) (2012). http://www.bkcase.org/grcse/, last accessed March 2013.

Graduate Software Engineering Reference Curriculum (GSwE) (2009). Available at http://www.gswe2009.org/, last accessed Feb. 2013.

Guide to the Systems Engineering Body of Knowledge (SEBoK) (2012). Available at http://www.sebokwiki.org/, last accessed March 2013.

* This exercise is suitable for a small research assignment.

Kilcay-Ergin, N. and Laplante, P.A. (2012). An online graduate requirements engineering course, *IEEE Transactions on Education*. Available at http://ieeexplore.ieee.org/stamp/stamp.jsp?tp = &arnumber = 6253277&isnumber = 4358717.

Laplante, P.A. and Neill, C. (2004). The demise of the waterfall model is imminent and other urban myths of software engineering, *ACM Queue*, 1(10): 10–15.

Laplante, P.A., Kalinowski, B., and Thornton, M. (2013). A principles and practices exam specification to support software engineering licensure in the United States of America, *Software Quality Professional*, 15(1): 4–15.

Marinelli, V. (2008). *An Analysis of Current Trends in Requirements Engineering Practice*, Master of Software Engineering Professional Paper, Malvern, PA: Penn State University, Great Valley School of Graduate Professional Studies.

Neill, C.J. and Laplante, P.A. (2003). Requirements engineering: The state of the practice, *Software*, 20(6): 40–45.

Rittel, H. and Webber, M. (1973). Dilemmas in a general theory of planning, *Policy Sciences*, 4: 155–169.

Royce, W. (1975). *Practical Strategies for Developing Large Software Systems*. Boston, MA: Addison-Wesley, 59.

Sikora, E., Tenbergen, B., and Pohl, K. (2012). Industry needs and research directions in requirements engineering for embedded systems, *Requirements Engineering*, 17: 57–78.

Software Engineering Body of Knowledge (SWEBOK) (2012). Available at http://www.computer.org/portal/web/swebok/html/contentsch2#ch2, last accessed Feb. 2013.

Sommerville, I. (2005). *Software Engineering*, 7th edition. Boston, MA: Addison-Wesley.

Voas, J. and Laplante, P. (2010). Effectively defining shall not requirements. *IT Professional*, 12(3): 46–53.

Wnuk, K., Regnell, B., and Berenbach, B. (2011). Scaling up requirements engineering—Exploring the challenges of increasing size and complexity in market-driven software development. In Berry, D. and X. French (Eds): *REFSQ 2011, LNCS 6606*, 54–59.

Zachman, J.A. (1987). A framework for information systems architecture, *IBM Systems Journal*, 26(3): 276–292.

Zave, P. (1997). Classification of research efforts in requirements engineering, *ACM Computing Surveys*, 29(4): 315–321.

Zave, P. and Jackson, M. (1997). Four dark corners of requirements engineering, *ACM Transactions on Software Engineering Methodology*, 6(1): 1–30.

Chapter 2

Preparing for Requirements Elicitation

Product Mission Statement

The first thing we need to do when undertaking the development of a new system, or redesign of an old one, is to obtain or develop a concise description of what it is supposed to do. Such a statement is often called a "product mission statement" (or "system mission statement"). Some organizations refer to a concept of operations (Conops) statement, which is similar to the mission statement, although the Conops tends to be longer. In some settings the Conops is a document resembling a very high-level requirements specification.

Section 1.1 of Appendix A contains a mission statement for the Smith smart home and Section 1 of Appendix B contains a Conops statement for the wastewater pumping control system.

The product mission statement or Conops acts as a focal point for all involved in the system, and it allows us to weigh the importance of various features by asking the question, "How does that functionality serve its intent?" In agile methodologies, discussed later, we could say that the mission statement or Conops plays the role of "system metaphor."

Writing mission statements can be contentious business, and many people resent or fear doing so because there can be a tendency to get bogged down on minutiae. Sometimes, product mission statements tend to get very long and, in fact, evolve into a Conops document. A product mission statement should be very short, descriptive, compelling, and never detailed.

One of the most widely cited "good" system mission statements is the one associated with the Starship *Enterprise* from the original *Star Trek* television series. That mission statement, roughly stated, is

> To explore strange new worlds, to seek out new life and new civilizations, to boldly go where no man has gone before.

This statement is clear, compelling, and inspiring. And it is "useful." Fans of this classic series will recall several episodes in which certain actions to be taken by the starship crew were weighed against the system mission statement.

So what might a product or system mission statement for the baggage handling system look like? How about:

> To automate all aspects of baggage handling from passenger origin to destination.

For the pet store POS system, consider:

> To automate all aspects of customer purchase interaction and inventory control.

These are not necessarily clever or awe-inspiring but they do convey the essence of the system. And they might be useful downstream when we need to calibrate the expectations of those involved in its specification. In globally distributed development in particular, the need for a system metaphor is of paramount importance.

Encounter with a Customer

Suppose your wife (or substitute "husband," "friend," "roommate," or whoever) asks you to go to the store to pick up the following items because she wants to make a cake:

- 5 pounds flour
- 12 large eggs
- 5 pounds sugar
- 1 pound butter

Off you go to the nearest convenience mart (which is close to your home). At the store, you realize that you are not sure if she wants white or brown sugar. So you call her from your cell phone and ask which kind of sugar she wants; you learn she needs brown sugar. You make your purchases and return home.

But your wife is unhappy with your selections. You bought the wrong kind of flour; she informs you that she wanted white and you bought wheat. You bought

the wrong kind of butter; she wanted unsalted. You brought the wrong kind of sugar too, dark brown; she wanted light brown. Now you are in trouble.

So you go back to the mart and return the flour, sugar, and butter. You find the white flour and brown sugar, but you could only find the unsalted butter in a tub (not sticks), but you assume a tub is acceptable to her. You make your purchase and return with the items. But now you discover that you made new mistakes. The light brown sugar purchase is fine, but the white flour you brought back is bleached; she wanted unbleached. And the butter in the tub is unacceptable; she points out that unsalted butter can be found in stick form. She is now very angry with you for your ignorance.

So, you go back to the store and sheepishly return the items, and pick up their proper substitutes. To placate your wife's anger, you also decide to buy some of her favorite chocolate candy.

You return home and she is still unhappy. Although you finally got the butter, sugar, and flour right, now your wife remembers that she is making omelets for supper, and that a dozen eggs won't be enough for the omelets and the cake; she needs 18 eggs. She is also not pleased with the chocolate; she informs you she is on a diet and that she doesn't need the temptation of chocolate lying around.

One more time you visit the mart and return the chocolate and the dozen eggs. You pick up 18 eggs and return home.

You think you have gotten the shopping right when she queries: "Where did you buy these things?" When you note that you bought the items at the convenience mart, she is livid. She feels the prices there are too high. You should have gone to the supermarket a few miles farther down the road.

We could go on and on with this example, with each time your wife discovering a new flaw in your purchases, changing her mind about quantity or brand, adding new items, subtracting others, and so on.

But what does this situation have to do with requirements engineering and stakeholders? The situation illustrates many points about requirements engineering. First, you need to understand the application domain. In this case, having a knowledge of baking would have informed you ahead of time that there are different kinds of butter, flour, and sugar and you probably would have asked focusing questions before embarking on your shopping trip. Another point from this scenario—customers don't always know what they want—your wife didn't realize that she needed more eggs until after you made three trips to the store. And there is yet one more lesson in the story: never make assumptions about what customers want. You thought that the tub butter was acceptable; it wasn't. You finally learned that even providing customers with more than they ask for (in this case her favorite chocolate) can sometimes be the wrong thing to do.

But in the larger sense, the most important lesson to be learned from this encounter with a customer is that they can be trouble. They don't always know what they want and, even when they do, they may communicate their wishes ineffectively. Customers can change their minds and they may have high expectations about what you know and what you will provide.

Because stakeholder interaction is so important, we devote the rest of this chapter to understanding the nature of stakeholders, and more specifically the stakeholders for whom the system is being built: the customers.

Stakeholders

Stakeholders represent a broad class of individuals who have some interest (a stake) in the success (or failure) of the system in question. For any system, there are many types of stakeholders, both obvious and sublime. The most obvious stakeholder of a system is the user.

We define the *user* as the class (consisting of one or more persons) who will use the system. The *customer* is the class (consisting of one or more persons) who is commissioning the construction of the system. Sometimes the *customer* is called the *client* (usually in the case of software systems) or *sponsor* (in the case where the system is being built not for sale, but for internal use). But in many cases the terms "customer," "client," and "sponsor" are used interchangeably depending on the context. Note that the sponsor and customer can be the same person. And often there is confusion between who the client is and who the sponsor is that can lead to many problems.

In any case, clients, customers, users, sponsors—however you wish to redefine these terms—are all stakeholders because they have a vested interest in the system. But there are more stakeholders than these. It is said that "The customer is always right, but there are more persons/entities with an interest in the system." In fact, there are many who have a stake in any new system. For example, typical stakeholders for any system might include:

- Customers (clients, users)
- The customers' customers (in the case of a system that will be used by third parties)
- Sponsors (those who have commissioned or will pay for the system)
- All responsible engineering and technical persons (e.g., systems, development, test, maintenance)
- Regulators (typically, government agencies at various levels)
- Third parties who have an interest in the system but no direct regulatory authority (e.g., standards organizations, user groups)
- Society (system safety)
- Environment (for physical systems)

And, of course, this is an incomplete list. For example, we could go on with a chain of customers' customers' customer . . ., where the delivered system is augmented by a third party, augmented again, delivered to a fourth party, and so on. This chain of custody has important legal implications too: when a system fails, who is responsible or liable for the failure? In any case, when we use the term "stakeholder," we need to remember that others, not just the customer, are involved.

Negative Stakeh

Negative stakeholde
These include comp
adversely affected, o
stakeholders, other
skeptical managers,
passive-aggressive
All negative stakeh
possible.

Finally, there a
who are, nonethele
may wield some po
ties include enviro
types, the self-inte
"gadflies," and the

Stakeholder I

It is very importa
(positive and neg
the requirements
Why do we start
holders and disco
being built in wh
stakeholders can
the system that

Stakeholder

One way to help

- Who is p
- Who is g
- Who is g
- What a
  aspect o
- What la
- Who is
  ing, ma
- Who w
- Who el
- Who el

Negative Stakeholders

Negative stakeholders are those who may be adversely affected by the system. These include competitors, investors, and people whose jobs will be changed, adversely affected, or displaced by the system. There are also internal negative stakeholders, other departments who will take on more workload, jealous rivals, skeptical managers, and more. These internal negative stakeholders can provide passive-aggressive resistance and create political nightmares for all involved. All negative stakeholders have to be recognized and accounted for as much as possible.

Finally, there are always individuals who are not directly affected by systems who are, nonetheless, interested in or opposed to those systems, and because they may wield some power or influence, they must be considered. These interested parties include environmentalists, animal activists, political zealots, advocates of all types, the self-interested, and so on. Some people call these kinds of individuals "gadflies," and they shouldn't be ignored.

Stakeholder Identification

It is very important to identify accurately and completely all possible stakeholders (positive and negative) for any system. Stakeholder identification is the first step the requirements engineer must take after the mission statement has been written. Why do we start with stakeholder identification? Imagine leaving out key stakeholders and discovering them later. Or worse, stakeholders discover that a system is being built in which they have an interest, and they have been ignored. These tardy stakeholders can try to impose all kinds of constraints and requirements changes to the system that can be very costly.

Stakeholder Questions

One way to help identify stakeholders is by answering the following set of questions:

- Who is paying for the system?
- Who is going to use the system?
- Who is going to judge the fitness of the system for use?
- What agencies (government) and entities (nongovernment) regulate any aspect of the system?
- What laws govern the construction, deployment, and operation of the system?
- Who is involved in any aspect of the specification, design, construction, testing, maintenance, and retirement of the system?
- Who will be negatively affected if the system is built?
- Who else cares if this system exists or doesn't exist?
- Who else?

Let's try this set of questions on the airline baggage handling system. These answers are not necessarily complete; over time, new stakeholders may be revealed. But by answering these questions as completely as we can now, we reduce the chances of overlooking a very important stakeholder late in the process.

- Who is paying for the system?
- Airline, grants, passengers, your tax dollars
- Who is going to use the system?
- Airline personnel, maintenance personnel, travelers (at the end)
- Who is going to judge the fitness of the system for use?
- Airline, customers, unions, FAA, OSHA, the press, independent rating agencies
- What agencies (government) and entities (nongovernment) regulate any aspect of the system?
- FAA, OSHA, union contracts, state and local codes
- What laws govern the construction, deployment, and operation of the system?
- Various state and local building codes, federal regulations for baggage handling systems, OSHA laws
- Who is involved in any aspect of the specification, design, construction, testing, maintenance, and retirement of the system?
- Various engineers, technicians, baggage handlers union, and the like
- Who will be negatively affected if the system is built?
- Passengers, union personnel
- Who else cares if this system exists or doesn't exist?
- Limousine drivers
- Who else?

And let's try this set of questions on the pet store POS system:

- Who is paying for the system?
- Pet store, consumers.
- Who is going to use the system?
- Cashiers, managers, customers (maybe if self-service provided). Who else?
- Who is going to judge the fitness of the system for use?
- Company execs, managers, cashiers, customers. Who else?
- What agencies (government) and entities (nongovernment) regulate any aspect of the system?
- Tax authorities, governing business entities, pet store organizations, Better Business Bureau. What else?
- What laws govern the construction, deployment, and operation of the system?
- Tax laws, business and trade laws? What else?
- Who is involved in any aspect of the specification, design, construction, testing, maintenance, and retirement of the system?
- Various engineers, CFO, managers, cashiers. We need to know them all.

- Who will be negatively affected if the system is built?
- Manual cash register makers, inventory clerks. Who else?
- Who else cares if this system exists or doesn't exist?
- Competitors, vendors of pet products. Who else?
- Who else?

Stakeholder/Customer Classes

Assuming that we have answered the stakeholder identification questions as completely as we can, the next step is to cluster these stakeholders into classes. Because many of the stakeholders we have identified are likely to be customers, and because the other stakeholders that we have identified are probably not going to be directly involved in the requirements elicitation process (we are not going to interview legislators, e.g.), we cluster as many stakeholders as possible into user classes.

The reason that we cluster the classes is that for economy we need to identify a small number of individuals with whom to have contact in order to represent the interests and desires of a large swath of customers/stakeholders. So, for each of the classes that we identify we'll select an individual or very small representative group for that class and deal only with those "champions." As for the other stakeholders, we will remain wary of their concerns (and legal implications) throughout the process, but we are not likely to consult with any representatives of those classes of stakeholder if we can avoid it.

We cluster customers into classes according to interests, scope, authorization, or other discriminating factors (in effect, these are equivalence classes). Then we select a champion or representative group for each user class. Next we select the appropriate technique(s) to solicit initial inputs from each class. This is the elicitation process discussed in the next chapter.

To illustrate the clustering of stakeholders/users into representative classes consider the baggage handling system. We might select the following representative classes:

- System maintenance personnel (who will be making upgrades and fixes)
- Baggage handlers (who interact with the system by turning it on and off, increasing/decreasing speed, and so on)
- Airline schedulers/dispatchers (who assign flights to baggage claim areas)
- Airport personnel (who reassign flights to different baggage claim areas)
- Airport managers and policy makers

For the pet store POS system we might have the following classes:

- Cashiers
- Managers
- System maintenance personnel (to make upgrades and fixes)

- Store customers
- Inventory/warehouse personnel (to enter inventory data)
- Accountants (to enter tax information)
- Sales department (to enter pricing and discounting information)
- Others

For each of these classes, we'll want to select a champion and deal with them during the elicitation process. We discuss this aspect further in the next chapter.

Stakeholder Characteristics

A requirements engineer should be well aware that individuals, even those within the same stakeholder class, may have individual characteristics that need to be taken into account. For example, in the baggage handling system the needs of elderly travelers are different from those of children or adults. Those with certain disabilities will have different needs. Non-English speakers will have other needs. In many cases it makes sense to divide stakeholder classes into subclasses according to these special needs and characteristics.

Moreover, communicating with individuals or groups within these subclasses probably requires the use of different techniques and most certainly enhanced empathy. Newell et al. (2006) provide a helpful overview of some of these challenges and their solution.

The differences in cultural characteristics also cannot be ignored when interacting with stakeholders from different countries. Seminal work by sociologist Geert Hofstede found that there are five dimensions along which cultural differences between countries could be perceived: comfort in dealing with authority (power distance), individualism, the tendency toward a masculine worldview (masculinity index), uncertainty avoidance, and long-term orientation. These differences matter to the requirements engineer. For example, an individual from a country with a high power distance may be unwilling to raise an important issue that could embarrass a superior. Or a female individual from a country with a high masculinity index may be afraid to speak up in a meeting when men are present. Therefore, different interaction techniques may be needed in these cases. An entertaining discussion of these issues can be found in Laplante (2010).

Finally, be aware of conflict of interest or "agency problems" as these can bias and obscure communications with stakeholders. That is, a member of one stakeholder class can be a member of another stakeholder class. For example, a cashier for the pet store POS system may also be a customer. So in communicating with this individual about how best to conduct a transaction, the requirements engineer cannot know whether the interests of a customer or a cashier are being more favorably represented. The requirements engineer should be aware of these potential biases and control for them in any communications with stakeholders.

Customer Wa

We mentioned that
what the customers
for "desired things"
ments engineer, sho
the admonition ab
you think are the
being said, it is alw
they can determi
This situation is e
than the requirem
cile wants and ne
good discussion
Jackson, and Dic

What Do Cu

The requirement
always easy to
exist on many l
petitive (it shou
the system's fea
they don't wan
tomers to set re

One way t
(1943) classic
human beings
(the bottom o
and spiritual.
level) if lower

Basic nee
physical safet
street to get f
up Maslow's
presumes tha
safety. You n
sacrifice the
you continu
members?)
to belong a
Roman cos
defined self

Customer Wants and Needs

We mentioned that the primary goal of the requirements engineer is to understand what the customers want. But discovering these wants or desiderata (from the Latin for "desired things") is challenging. You also might think that you, as the requirements engineer, should suggest to the customer what he or she needs, but remember the admonition about substituting your own value system for someone else's: what you think are the customers' needs might be something that they don't want. That being said, it is always helpful to reveal new functionality to the customers so that they can determine if there are features that they need but had not considered. This situation is especially relevant when the customer has less domain knowledge than the requirements engineer. At some point, however, you will need to reconcile wants and needs; requirements prioritization will be very useful then. A very good discussion of stakeholder engagement to identify desiderata is given by Hull, Jackson, and Dick (2011).

What Do Customers Want?

The requirements engineer seeks to satisfy customer wants and needs, but it is not always easy to know what these are. Why? Because customers' wants and needs exist on many levels: practical (e.g., minimum functionality of the system), competitive (it should be better than brand X), selfish (they want to show off and tout the system's features), and more. And sometimes the customers "want it all" and they don't want to overpay. Requirements engineers, therefore, have to help customers to set realistic goals for the system to be built.

One way to understand the needs levels of customers is to revisit Maslow's (1943) classic hierarchy of self-actualization (Figure 2.1). Maslow theorized that human beings will seek to satisfy (actualize) their needs starting with the most basic (the bottom of the pyramid) and work their way up to the most elusive, esoteric, and spiritual. But they will never move very far up the pyramid (or stay at a higher level) if lower levels of needs/wants are not satisfied first.

Basic needs include food, water, and sleep. These take precedence over one's physical safety; however, if you were starving, you would risk crossing a very busy street to get food on the other side. People risk jail time by stealing bread. Higher up Maslow's pyramid is the need to be loved and to belong to some group, but he presumes that these needs are fundamentally subordinated to the need for physical safety. You might argue about this one, but the thinking is that some people will sacrifice the chance for love in order to preserve their physical well-being. (Would you continue belonging to the Sky Diving Club just because you liked one of its members?) Next, one's self-esteem is important, but not as important as the need to belong and be loved (which is why you will humiliate yourself and dress in a Roman costume for your crazy sister-in-law's wedding). Finally, Maslow (1943) defined self-actualization as "man's desire for fulfillment, namely to the tendency

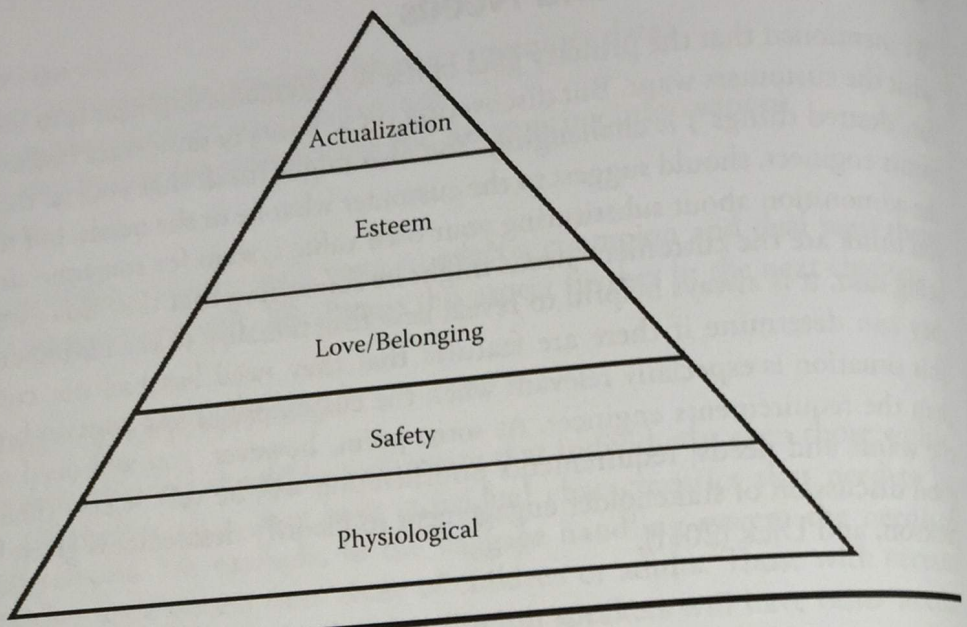

Figure 2.1 Maslow's hierarchy.

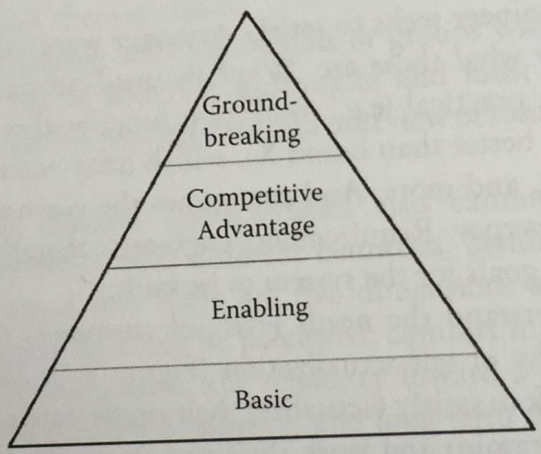

Figure 2.2 Hierarchy of customer needs/wants.

for him to become actually in what he is potentially: to become everything that one is capable of becoming."

A variation of Maslow's hierarchy, depicted in Figure 2.2, can help explain the needs and wants of customers. Here the lowest level is basic functionality. Being a point of sale system implies certain functions must be present, such as create a sale, return an item, update inventory, and so on. At the enabling level, the customer desires features that provide enabling capabilities with respect to other systems (software, hardware, or process) within the organization. So the POS system ties into some management software that allows for real-time sales data to be tracked by managers for forecasting or inventory control purposes. The functionality at the enabling level may not meet or exceed competitors' capabilities. Those functional needs are met at the competitive advantage level. Here the customer wishes for

this new system to provide capabilities that exceed those of the competition or otherwise create a business advantage. Finally, groundbreaking desires would imply development of technology that exceeds current theory or practice and has implications and applications beyond the system in question. For example, some kind of new data-mining technology might be desired that exceeds current technologies. As with the Maslow hierarchy, the idea is that the lower-level functionality must not be sacrificed to meet the higher levels of functionality.

Although this hierarchy implies four levels of importance of need, it is likely that in any situation there can be more or fewer levels. But the basic idea is to discover and organize customers' needs according to some meaningful hierarchy, which will be most helpful with requirements prioritization later. And this is not the first time that Maslow's theory was used to explain the needs of customers or users. For example, Valacich, Parboteeah, and Wells (2007) described a modified four-level Maslow hierarchy to explain user preferences in web-based user interfaces.

In any case, let's return to our examples to consider some of the wants for the baggage handling and pet store POS systems.

For an airline baggage handling system customers probably want:

- Safety
- Speed
- Reliability
- Fault-tolerance (no broken luggage!)
- Maintainability
- And so on

For the pet store POS system, customers want:

- Speed
- Accuracy
- Clarity (in the printed receipt)
- Efficiency
- Ease of use (especially if self-service provided)
- And more

So we would use our best efforts to attend to these needs. The problem becomes, however, how do we measure satisfaction of these needs? Because, if these wants and desires cannot be measured, then we will never achieve them. We discuss the issue of measurable requirements in Chapter 5.

What Don't Customers Want?

Sometimes customers are very explicit in what they don't want the system to do. These specific undesirable features or "do not wants" or "shall not" requirements

are frequently overlooked by the requirements engineer. Unwanted features can include:

- Undesirable performance characteristics
- Aesthetic features
- Goldplating (excessive and unnecessary features)
- Safety concerns (hazards)

The "shall not" requirements are often the hardest to capture. Sometimes customers don't know what they don't want until they see it. For example, upon seeing the delivered system (or a prototype), they exclaim: "I know I said I wanted it to do that, but I guess I really didn't mean that."

For illustrative purposes, here are some examples of unwanted features of the baggage handling system:

- The system shall not immediately shut down if main airport power is lost.
- The system shall not cause a failure in the main airline computer system.
- The system shall not cause baggage to be destroyed at a rate higher than 1 bag per minute.

You can see how hard it is to describe what the system is not supposed to do. We study the issue of unwanted behavior later. In the meantime, to illustrate the point, here are some "shall not" requirements for the pet store POS system:

- If the register tape runs out, the system shall not crash.
- If a product code is not found, the system shall not crash.
- If a problem is found in the inventory reconciliation code, the current transaction shall not be aborted.

Many of the elicitation techniques that we discuss in the next chapter will tend to uncover unwanted features, but the requirements engineer should always try to discover what the customer does not want as vigorously as what the customer does want.

Why Do Customers Change Their Minds?

One of the greatest challenges in dealing with stakeholders, especially customers, is that they sometimes don't know precisely what they want the system to do. Why is this so? First, no one is omnipotent. Customers can't see every possible desideratum, unless they want a complete replica of another system. Helping the customer find these new features as early as possible is the job of the requirements engineer.

Another reason why customers change their minds is that, with varying levels of requirements (e.g., features, constraints, business rules, quality attributes, etc.),

what is important might
change during the system

Sometimes the envir
ers operate changes (phy
changes, etc.). For exam
rules about pet toys mi
event might require a si

Some requirements
them. For example, in
POS system? The custo
the system designer tho

Another reason for
return on investment.
customer deemed the
add substantial econo
menting the requirem
paperwork that shoul
driver. Prioritization a
ment data are particul

Sometimes the cu
because that is the wa
is (you need to educa
requirements enginee
(how much is up to y

Finally, customer
(e.g., the informatio
don't think you will
later in the project a

Stakeholder P

Up until now, we h
holder but, of cours
For example, the c
not be as importan
baggage handling s
of the customer; f
standards.

Because we ha
conflict, we rank
tions. Usually ran
requirements shou

what is important might change as these "nonrequirements" (*e.g., business rules*) change during the system life cycle.

Sometimes the environment in which the system functions and the customers operate changes (physical changes, economic changes, competition, regulatory changes, etc.). For example, while building the pet store POS system, state taxation rules about pet toys might change (maybe only squeaky toys are taxed), and this event might require a significant redesign of the taxation portion of the code.

Some requirements are so "obvious" that the customer doesn't think to stipulate them. For example, in what order do you print a list of customers in the pet store POS system? The customer's "obvious" answer is alphabetically by last name, but the system designer thought numerically by customer ID.

Another reason for changing requirements has to do with the discovery of the return on investment. In one project, for example, it was discovered that what the customer deemed the most important requirement (paperwork reduction) did not add substantial economic value to the business with respect to the cost of implementing the requirement. It turned out that a secondary requirement, that the paperwork that should be eliminated had to be nontrivial, was the key economic driver. Prioritization and avoiding late changes is one reason why return on investment data are particularly useful in driving the requirements set.

Sometimes the customers are simply flighty or fickle. They change their minds because that is the way they operate. Or they simply don't know what a requirement is (you need to educate them). Other customers are simply, well, stupid. As their requirements engineer you have to tolerate a certain amount of this changeability (how much is up to you).

Finally, customers will deliberately withhold information for a variety of reasons (e.g., the information is proprietary, they distrust you, they don't like you, they don't think you will understand). The information withheld can rear its ugly head later in the project and require costly changes to the system.

Stakeholder Prioritization

Up until now, we have mostly been referring to the customer as the primary stakeholder but, of course, there are others. Not all stakeholders are of equal importance. For example, the concerns of the baggage handlers union are important, but may not be as important as the airport authority (the customer) who is paying for the baggage handling system. On the other hand, federal regulations trump the desires of the customer; for example, the system must comply with all applicable federal standards.

Because we have many stakeholders and some of their needs and desires may conflict, we rank or prioritize the stakeholder classes to help resolve these situations. Usually rank denotes the risk of not satisfying the stakeholder (e.g., legal requirements should be #1 unless you want to go to jail). Ranking the stakeholders

will lead to requirements prioritization, which is the key to reconciliation and risk mitigation, so get used to it.

Table 2.1 contains a partial list of stakeholders for the baggage handling system ranked in a simple high, medium, and low priority scheme. A rationale for the rank assignment is included.

Table 2.2 contains a partial list of stakeholders for the pet store POS system ranked using ratings 1 through 6, where 1 represents highest importance or priority.

Table 2.1 Partial Ranking of Stakeholders for the Baggage Handling System

Stakeholder Class	*Rank*	*Rationale*
System maintenance personnel	Medium	They have moderate interaction with the system.
Baggage handlers	Medium	They have regular interaction with the system but have an agenda that may run counter to the customer.
Airline schedulers/ dispatchers	Low	They have little interaction with the system.
Airport personnel	Low	Most other airport personnel have little interaction with the system.
Airport managers and policy makers ("the customer")	High	They are paying for the system.

Table 2.2 Partial Ranking of Stakeholders for the Pet Store POS System

Stakeholder Class	*Rank*	*Rationale*
Cashiers	2	They have the most interaction with the system.
Managers	1	They are the primary customer/sponsor.
System maintenance personnel	4	They have to fix things when they break.
Store customers	3	They are the most likely to be adversely affected.
Inventory/warehouse personnel	6	They have the least direct interaction with the system.
Accountants/sales personnel	5	They read the reports.

You can certainly argue with this ranking and prioritizations; for example, you may think that the store customer is the most important person in the POS system. But this disagreement highlights an important point: it is early in the requirements engineering process when you want to argue about stakeholder conflicts and prioritization, not later when design decisions may have already been made that need to be undone.

Communicating with Customers and Other Stakeholders

One of the most important activities of the requirements engineer is to communicate with customers, and at times with other stakeholders. In many cases, aside from the sales personnel, the requirements engineer is the customer-facing side of the business. Therefore, it is essential that all communications be conducted clearly, ethically, consistently, and in a timely fashion.

The question arises as to what the best format is for communications with customers. There are many ways to communicate, and each has specific advantages and disadvantages. For example, in-person meetings are very effective. Verbal information is conveyed via the language used, but also more subtle clues can be conveyed from voice quality, tone and inflection, and body language. In fact, agile software methodologies advocate having a customer representative on site at all times to facilitate continuous in-person communications. Agile methodologies for requirements engineering are discussed in Chapter 7. But in-person meetings are not economical and they consume a great deal of time. Furthermore, when you have multiple customers, even geographically distributed customers, how do you meet with them? Together? Separately? Via teleconference? All of these questions have to be considered.

Well-planned, intensive group meetings can be an effective form of communication for requirements engineering, and we discuss such techniques in the next chapter. But these meetings are expensive and time consuming, and can disrupt the client's business.

Providing periodic status reports to customers during the elicitation and beyond can help to avoid some of these problems. At least from a legal standpoint, the requirements engineer has been making full disclosure of what he knows and what he does not.

Should written communications with the customer take the form of legal contracts and memoranda? The advantage of formal contracts (or change request notices) is that this kind of communication can avoid disputes, or at least provide evidence in the event of a dispute. After all, any communication with the customer can be relevant in court. But formal communications are impersonal, can slow the process of requirements engineering significantly, and can be costly (especially if a lawyer is involved).

Telephone or teleconference calls can be used to communicate throughout the requirements engineering process. The informality and speed of this mode are highly desirable. But even with teleconferencing, some of the nuance of co-located communication is lost, and there are always problems of misunderstanding, dropped calls, and interruptions. And the informality of the telephone call is also a liability: every communication with a customer has potential legal implications but it is usually inconvenient to record every call.

E-mail can be effective as a means of communication, and its advantages and disadvantages fall somewhere between written memoranda and telephone calls. E-mail is both spontaneous and informal, but it is persistent: you can save every e-mail transaction. But as with telephone calls, some of the interpersonal nuanced communication is lost and, as a legal document, e-mail trails are less convincing than formal change request notices.

Finally, Wiki technology can and has been used to communicate requirements information with customers and other stakeholders. The Wiki can be used as a kind of whiteboard on which ideas can be shared and refined. Furthermore, with some massaging, the Wiki can be evolved into the final software requirements specification document. And there are ways to embed executable test cases into the SRS itself using the FitNesse acceptance testing framework (fitnesse.org). These issues are explored further in Chapter 8.

Managing Expectations

A big part of communicating with customers is managing expectations. Expectations really matter in all endeavors, not just requirements engineering. If you don't believe this fact, consider the following situations:

> **Situation A:** Imagine you were contracted to do some work as a consultant and you agreed to a fee of $5,000 for the work. You complete the work and the customer is satisfied. But your client pays you $8,000. He has had a successful year and he wants to share the wealth. How do you feel?
>
> **Situation B:** Now reset the clock; imagine the previous situation didn't happen yet. Imagine now that you agree to do the same work as before, but this time for $10,000. You do exactly the same amount of work as you did in situation A and the customer is satisfied. But now the customer indicates that he had a very bad year and that all he can pay is $8,000, take it or leave it. How do you feel?

In both situation A and situation B you did exactly the same amount of work and you were paid exactly the same amount of money. But you would be ecstatic in situation A, and upset in situation B. Why? What is different?

The difference is in your expectations. In situation A you expected to be paid $5,000 but the customer surprised you and exceeded your expectations, making you happy. In situation B your expectations of receiving $10,000 were not met, making you unhappy.

Some might argue that, in any endeavor, this example illustrates that you should set customers' expectations low deliberately and then exceed them, so that you can make the customer extremely happy. But this will not always work, and certainly will not work in the long run; people who get a reputation for padding schedules or otherwise low-balling expectations lose the trust of their customers and clients.*

Also recognize that you exert tremendous conscious and unconscious influence on stakeholders. When communicating with customers avoid saying such words as, "I would have the system do this …" or "I don't want the system to do that …." These phrases may influence the customer to make a decision that will be regretted later and potentially blamed on you.

Therefore, our goal as requirements engineers is to manage customers' expectations carefully; that is, understand, adjust, monitor, reset, and then meet customer expectations at all times.

Stakeholder Negotiations

It is inevitable that along the way the requirements engineer must negotiate with customers and other stakeholders. Often the negotiations deal with convincing the customer that some desired functionality is impossible or they deal with the concerns of other stakeholders. And expectation setting and management throughout the life cycle of any system project is an exercise in negotiation. Although we are not about to embed a crash course in negotiation theory in this book, we wanted to mention a few simple principles that should be remembered.

Set the ground rules up front. When a negotiation is imminent, make sure that the scope and duration of the discussions are agreed. If there are to be third parties present, make sure that this is understood. If certain rules need to be followed, make both parties aware. Trying to eliminate unwanted surprises for both sides in the negotiation will lead to success.

Understand people's expectations. Make sure you realize that what matters to you might not matter to the other party. Some people care about money; others care more about their image, reputation, or feelings. When dealing with negotiations surrounding system functionality, understand what is most important to the customer. Ranking requirements will be most helpful in this regard.

---

* The author often jokes when giving seminars that "I like to set expectations deliberately low … and then meet them."

Look for early successes. It always helps to build positive momentum if agreement, even on something small, can be reached. Fighting early about the most contentious issues will amplify any bad feelings and make agreement on those small issues more difficult later.

Be sure to give a little and push back a little. If you give a little in the negotiation, it always demonstrates good faith. But the value of pushing back in a negotiation is somewhat counterintuitive. It turns out that by not pushing back, you leave the other party feeling cheated and empty.

For example, suppose someone advertises a used car for sale at $10,000. You visit the seller, look at the car, and offer $8,000. The seller immediately accepts. How do you feel? You probably feel that the seller accepted too easily, and that he had something to hide. Or, you feel that the $10,000 asking price was grossly inflated.If you had offered $10,000, the seller would have accepted that. How greedy of him! You would have actually felt better if the seller refused your offer of $8,000 but countered with $9,000 instead. So, push back a little.

Conclude negotiating only when all parties are satisfied. Never end a negotiation with open questions or bad feelings. Everyone needs to feel satisfied and whole at the end of a negotiation. If you do not ensure mutual satisfaction, you will likely not do business together again, and your reputation at large may be damaged (customers talk to one another).

There are many good texts on effective negotiation (e.g., Cohen 2000), and it is advisable that all requirements engineers continuously practice and improve their negotiating skills.

Uncovering Stakeholder Goals

From the "encounter with a customer" it should be clear that understanding customer goals is critical to the success of the requirements engineering process. Goals further detail the intentions of the system summarized in the product mission statement.

For example, some goals for pet store POS might be:

- Provide "hassle-free" shopping for all customers.
- Support all coupon and discount processing.
- Support all customer loyalty programs.
- Fully automate inventory entry and maintenance.
- Support all local, state, and federal tax processing.

The relationship between the product mission statement, goals, and requirements is depicted in Figure 2.3. Notice how the goals provide further articulation of the intent contained in the product mission statement. Those goals are operationally described and detailed through the requirements, which must be verified through measurements.

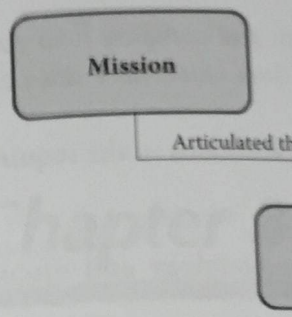

Figure 2.3 Goal-base

Goal-oriented requ
goals in order to obtain
goal-oriented requirem
These approaches aim
requirements (Asnar, C

Because each stake
must undertake goal
zation so that we can
goal-based understand
Chapter 3.

Exercises

2.1 Why is it imp
requirements
2.2 What is the
company's m
2.3 When should
2.4 Under what
considered s
2.5 Think of a sy
statement is

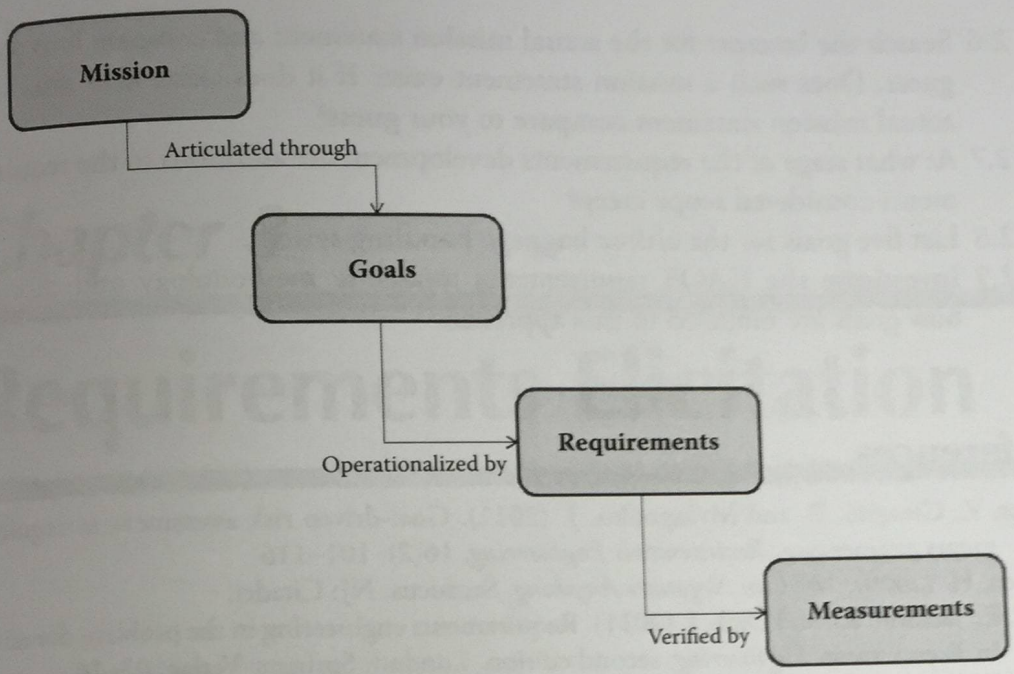

Figure 2.3 Goal-based requirements engineering.

Goal-oriented requirements engineering involves the analysis of stakeholder goals in order to obtain new functional requirements to meet these goals. Existing goal-oriented requirements engineering techniques include KAOS, i*, and Tropos. These approaches aim at modeling the "who, what, why, where, when, and how" of requirements (Asnar, Giorgini, and Mylopoulos 2011).

Because each stakeholder may have a different set of goals for any system, we must undertake goal understanding after stakeholder identification and prioritization so that we can reconcile differences in goal sets. A simple technique for goal-based understanding using the goal-question-metric paradigm is discussed in Chapter 3.

Exercises

2.1 Why is it important to have a Conops or mission statement at the start of requirements engineering?
2.2 What is the relationship of a system or product mission statement to a company's mission statement?
2.3 When should a domain vocabulary be established?
2.4 Under what circumstances might the customer's needs and desires be considered secondary?
2.5 Think of a system or product you use often and try to guess what the mission statement is for that product, or the company that makes it.

2.6 Search the Internet for the actual mission statement and compare it to your guess. Does such a mission statement exist? If it does exist, how does the actual mission statement compare to your guess?

2.7 At what stage of the requirements development are additions to the requirements considered scope creep?

2.8 List five goals for the airline baggage handling system.

*2.9 Investigate the KAOS requirements modeling methodology and discuss how goals are modeled in this approach.

References

Asnar, Y., Giorgini, P., and Mylopoulos, J. (2011). Goal-driven risk assessment in requirements engineering. *Requirements Engineering*, 16(2): 101–116.

Cohen, H. (2000). *You Can Negotiate Anything*, Secaucus, NJ: Citadel.

Hull, E., Jackson, K., and Dick, J. (2011). Requirements engineering in the problem domain. In *Requirements Engineering*, second edition. London: Springer-Verlag, 93–14.

Laplante, P.A. (2010). Where in the world is Carmen Sandiego (and is she a software engineer)? *IT Professional*, 12(6): 10–13.

Maslow, A. (1943). A theory of human motivation. *Psychological Review*, 50: 370–396.

Newell, A.F., Dickinson, A., Smith, M.J., and Gregor, P. (2006). Designing a portal for older users: A case study of an industrial/academic collaboration. *ACM Transactions on Computer-Human Interaction*. 13(3): 347–375.

Valacich, J.H., Parboteeah, D.V., and Wells, J.D. (2007). The online consumer's hierarchy of needs. *Communications of the ACM*, 50(9): 84–90.