CHAPTER 1

Requirements Management

This chapter starts by defining the concepts of requirements and stakeholders. Then we describe what types of requirements can exist in a project. The relationships between these requirements are presented in the form of a pyramid. The concept of traceability is introduced (which requirement is derived from which). Characteristics of a good requirement are presented. Examples of problematic requirements are given, together with some guidelines on how to fix them. General steps in requirements management (RM) during the project lifecycle are shown. The main steps navigate through the requirements pyramid from top to bottom.

1.1 Definition of a Requirement and a Stakeholder

A requirement is defined as "a condition or capability to which a system must conform." It can be any of the following:

- A capability needed by a customer or user to solve a problem or achieve an objective
- A capability that must be met or possessed by a system to satisfy a contract, standard, specification, regulation, or other formally imposed document
- A restriction imposed by a stakeholder

Let's define a concept of a stakeholder because this word occurs many times in this book. Usually the stakeholder is defined as someone who is affected by the system that is being developed. The two main types of stakeholders are users and customers. Users are people who will be using the system. Customers are the people who request the system and are responsible for approving it. Usually customers pay for the development of the system. It is important to distinguish between these two groups of stakeholders because sometimes requirements provided by both groups conflict. In most of these types of conflicts, customer requests take precedence over

user requests. In the travel agency website example used in this book, a customer is a travel agency owner, and the users are all the people who will be using this website through the Internet. Besides customers and users, many other types of stakeholders cannot be neglected.

For the purposes of this book, we will call the stakeholder anybody involved in the system (either during development or after it is completed) and anybody who may have any requirement for the system.

Here are some of the people who may be considered stakeholders:

- Anyone participating in the development of the system (business analysts, designers, coders, testers, project managers, deployment managers, use case designers, graphic designers)
- Anyone contributing knowledge to the system (domain experts, authors of documents that were used for requirements elicitation, owners of the websites to which a link is provided)
- Executives (the president of the company that is represented by customers, the director of the IT department of the company that designs and develops the system)
- People involved in maintenance and support (website hosting company, help desk)
- Providers of rules and regulations (rules imposed by search engines regarding content of the website, government rules, state taxation rules)

1.2 Requirements Pyramid

Depending on the format, source, and common characteristics, the requirements can be split into different requirement types. Here are some requirement types that are often used in projects:

- Stakeholder need: a requirement from a stakeholder
- Feature: a service provided by the system, usually formulated by a business analyst; a purpose of a feature is to fulfill a stakeholder need
- Use case: a description of system behavior in terms of sequences of actions
- Supplementary requirement: another requirement (usually nonfunctional) that cannot be captured in use cases
- Test case: a specification of test inputs, execution conditions, and expected results
- Scenario: a specific sequence of actions; a specific path through a use case

These requirement types can be presented in the form of a pyramid, as shown in Figure 1.1.

At the top level are stakeholder needs. On the lower levels are features, use cases, and supplementary requirements. Quite often, at different levels of these requirements, different levels of detail are captured. The lower the level, the more detailed the requirement. For example, a need might be “Data should be persistent.” The feature can refine this requirement to be “System

should use a relational database." On the supplementary specification level, the requirement is even more specific: "System should use Oracle 9i database." The further down, the more detailed the requirement. One of the best practices of requirements management is to have at least two different levels of requirement abstraction. For example, the Vision contains high-level requirements (features), and the lower levels in the pyramid express the requirements at a detailed level. Senior stakeholders (such as vice presidents) do not have time to read 200 pages of detailed requirements but should be expected to read a 12-page Vision document.

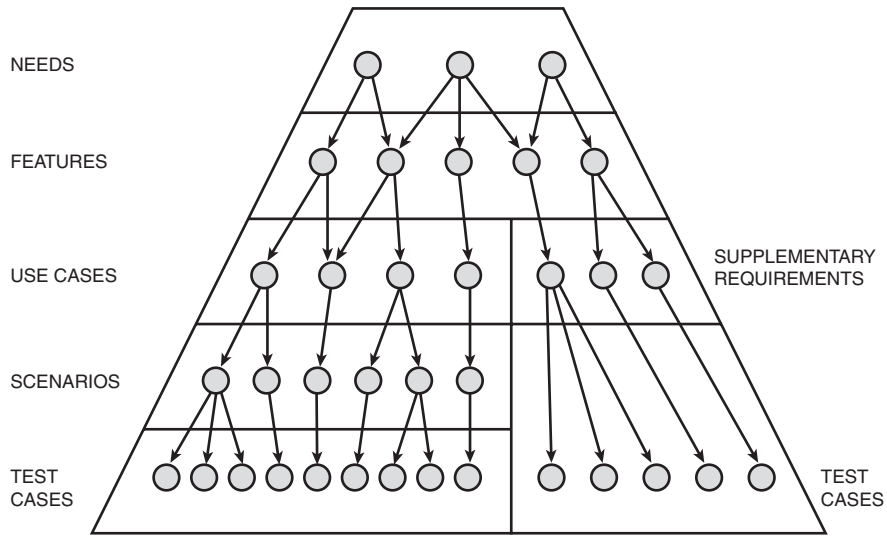

Figure 1.1 The requirements pyramid.

However, it is up to business analysts to decide on the granularity of requirements at each level. Nothing is wrong with placing quite detailed requirements from stakeholders on the stakeholder needs level.

The main difference between needs and features is in the source of the requirement. Needs come from stakeholders, and features are formulated by business analysts.

The role of test cases is to check if use cases and supplementary requirements are implemented correctly. Scenarios help derive use cases from test cases and facilitate the design and implementation of specific paths through use cases. In RequisitePro we can define many other requirement types, such as glossary terms and actors. They are not pure requirements conforming to the definition provided at the beginning of this chapter, but if we represent them in RequisitePro as requirements, we gain flexibility to track their attributes and traceability using same mechanisms that are provided for other requirement types.

1.3 Traceability between Requirements

Traceability is a technique that provides a relationship between different levels of requirements in the system. This technique helps you determine the origin of any requirement. Figure 1.1 illustrates how requirements are traced from the top level down. Every need usually maps to some features. Generally, it is a many-to-many relationship because one need can trace to many features, but one feature may be derived from many needs. One need mapping to one feature is also a common case. The features map to use cases in a many-to-many relationship. The features also map to supplementary requirements in a many-to-many relationship.

Every use case maps to one or more scenarios, so a one-to-many relationship exists between use cases and scenarios. Scenarios map to test cases in a one-to-many relationship.

Traceability plays several important roles:

- Verifying that an implementation fulfills all requirements: Everything that the customer requested was implemented.
- Verifying that the application does only what was requested: Don't implement something that the customer never asked for.
- Impact analysis: What elements will be affected when we consider adding a new requirement or changing an existing one?
- Helping with change management: When some requirements change, we want to know which test cases should be redone to test this change.

A traceability item is a project element that needs to be traced from another element. In terms of RequisitePro, it's everything that is represented by an instance of the requirement type. Some examples of requirement types in RequisitePro are stakeholder needs, features, use cases, actors, and glossary terms. RequisitePro has a convenient way of showing traceability in special views.

1.4 Characteristics of a Good Requirement

A requirement needs to meet several criteria to be considered a "good requirement" [HUL05] [LEF03] [LUD05][YOU01]. Good requirements should have the following characteristics:

- Unambiguous
- Testable (verifiable)
- Clear (concise, terse, simple, precise)
- Correct
- Understandable
- Feasible (realistic, possible)
- Independent
- Atomic
- Necessary
- Implementation-free (abstract)

Besides these criteria for individual requirements, three criteria apply to the set of requirements. The set should be

- Consistent
- Nonredundant
- Complete

The sample project used in this book is an online travel agency, as shown in Figure 1.2. You're probably familiar with this type of application because variations of it can be found on several websites. The project is complex enough to show possible relationships between various requirements types, but it is small enough to be easily understood. Most of the examples in this chapter (and the other chapters) are related to this project.

Figure 1.2 The home page of an online travel agency.

Let's discuss each of the criteria of a good requirement and show some examples.

Unambiguous

There should be only one way to interpret the requirement. Sometimes ambiguity is introduced by undefined acronyms:

REQ1 The system shall be implemented using ASP.

Does ASP mean Active Server Pages or Application Service Provider? To fix this, we can mention a full name and provide an acronym in parentheses:

REQ1 The system shall be implemented using Active Server Pages (ASP).

Here's another example:

REQ1 The system shall not accept passwords longer than 15 characters.

It is not clear what the system is supposed to do:

- The system shall not let the user enter more than 15 characters.
- The system shall truncate the entered string to 15 characters.
- The system shall display an error message if the user enters more than 15 characters.

The corrected requirement reflects the clarification:

REQ1 The system shall not accept passwords longer than 15 characters. If the user enters more than 15 characters while choosing the password, an error message shall ask the user to correct it.

Some ambiguity may be introduced through the placement of a certain word:

REQ1 On the "Stored Flight" screen, the user can only view one record.

Does this mean that the user can "only view," not delete or update, or does it mean that the user can view *only one* record, not two or three?

One way to fix the problem is to rewrite the requirement from the system's point of view:

REQ1 On the "Stored Flight" screen, the system shall display only one flight.

Testable (Verifiable)

Testers should be able to verify whether the requirement is implemented correctly. The test should either pass or fail. To be testable, requirements should be clear, precise, and unambiguous. Some words can make a requirement untestable [LUD05]:

- Some adjectives: robust, safe, accurate, effective, efficient, expandable, flexible, maintainable, reliable, user-friendly, adequate
- Some adverbs and adverbial phrases: quickly, safely, in a timely manner
- Nonspecific words or acronyms: etc., and/or, TBD

Such a requirement might look something like this:

REQ1 The search facility should allow the user to find a reservation based on Last Name, Date, etc.

In this requirement, all search criteria should be explicitly listed. The designer and developer cannot guess what the user means by "etc."

Other problems can be introduced by ambiguous words or phrasing:

- Modifying phrases: as appropriate, as required, if necessary, shall be considered
- Vague words: manage, handle
- Passive voice: the subject of the sentence receives the action of the verb rather than performing it

  REQ1 The airport code shall be entered by the user.

  REQ2 The airport code shall be entered.

The first example shows a classic example of passive voice. In active voice it would read "The user shall enter the airport code." As the second example shows, another result of the use of passive voice is that the agent performing the action is sometimes omitted. Who should enter this code—the system or the user?

- Indefinite pronouns: few, many, most, much, several, any, anybody, anything, some, somebody, someone, etc.

  REQ1 The system shall resist concurrent usage by many users.

What number should be considered "many"—10, 100, 1,000?

Clear (Concise, Terse, Simple, Precise)

Requirements should not contain unnecessary verbiage or information. They should be stated clearly and simply:

> *REQ1 Sometimes the user will enter Airport Code, which the system will understand, but sometimes the closest city may replace it, so the user does not need to know what the airport code is, and it will still be understood by the system.*

This sentence may be replaced by a simpler one:

> *REQ1 The system shall identify the airport based on either an Airport Code or a City Name.*

Correct

If a requirement contains facts, these facts should be true:

> *REQ1 Car rental prices shall show all applicable taxes (including 6% state tax).*

The tax depends on the state, so the provided 6% figure is incorrect.

Understandable

Requirements should be grammatically correct and written in a consistent style. Standard conventions should be used. The word "shall" should be used instead of "will," "must," or "may."

Feasible (Realistic, Possible)

The requirement should be doable within existing constraints such as time, money, and available resources:

> *REQ1 The system shall have a natural language interface that will understand commands given in English language.*

This requirement may be not feasible within a short span of development time.

Independent

To understand the requirement, there should not be a need to know any other requirement:

> *REQ1 The list of available flights shall include flight numbers, departure time, and arrival time for every leg of a flight.*
>
> *REQ2 It should be sorted by price.*

The word "It" in the second sentence refers to the previous requirement. However, if the order of the requirements changes, this requirement will not be understandable.

Atomic

The requirement should contain a single traceable element:

> *REQ1 The system shall provide the opportunity to book the flight, purchase a ticket, reserve a hotel room, reserve a car, and provide information about attractions.*

This requirement combines five atomic requirements, which makes traceability very difficult. Sentences including the words "and" or "but" should be reviewed to see if they can be broken into atomic requirements.

Necessary

A requirement is unnecessary if

- None of the stakeholders needs the requirement.

or

- Removing the requirement will not affect the system.

An example of a requirement that is not needed by a stakeholder is a requirement that is added by developers and designers because they assume that users or customers want it. For example, the fact that a developer thinks that users would like a feature that displays a map of the airport and he knows how to implement it is not a valid reason to add this requirement.

An example of a requirement that can be removed because it does not provide any new information might look like the following:

> *REQ1 All requirements specified in the Vision document shall be implemented and tested.*

Implementation-free (Abstract)

Requirements should not contain unnecessary design and implementation information:

> *REQ1 Content information shall be stored in a text file.*

How the information is stored is transparent to the user and should be the designer's or architect's decision.

Consistent

There should not be any conflicts between the requirements. Conflicts may be direct or indirect. Direct conflicts occur when, in the same situation, different behavior is expected:

REQ1 Dates shall be displayed in the mm/dd/yyyy format.

REQ2 Dates shall be displayed in the dd/mm/yyyy format.

Sometimes it is possible to resolve the conflict by analyzing the conditions under which the requirement takes place. For example, if REQ1 was submitted by an American user and REQ2 by a French user, the preceding requirements may be rewritten as follows:

REQ1 For users in the U.S., dates shall be displayed in the mm/dd/yyyy format.

REQ2 For users in France, dates shall be displayed in the dd/mm/yyyy format.

This can eventually lead to the following requirement:

REQ3 Dates shall be displayed based on the format defined in the user's web browser.

Another example of a direct conflict can be seen in these two requirements:

REQ1 Payment by PayPal shall be available.

REQ2 Only credit card payments shall be accepted.

In this case the conflict cannot be resolved by adding conditions, so one of the requirements should be changed or removed.

Indirect conflict occurs when requirements do not describe the same functionality, but it is not possible to fulfill both requirements at the same time:

REQ1 System should have a natural language interface.

REQ2 System shall be developed in three months.

Some requirements do not conflict, but they use inconsistent terminology:

REQ1 For outbound and inbound flights, the user shall be able to compare flight prices from other, nearby airports.

REQ2 The outbound and return flights shall be sorted by the smallest number of stops.

To describe the same concept, in the first requirement the term "inbound flights" is used, and in the second requirement the term "return flights" is used. The usage should be consistent.

Nonredundant

Each requirement should be expressed only once and should not overlap with another requirement:

REQ1 A calendar shall be available to help with entering the flight date.

REQ2 The system shall display a pop-up calendar when entering any date.

The first requirement (related to only the flight date) is a subset of the second one (related to any date entered by the user).

Complete

A requirement should be specified for all conditions that can occur:

REQ1 *A destination country does not need to be displayed for flights within the U.S.*

REQ2 *For overseas flights, the system shall display a destination country.*

What about flights to Canada and Mexico? They are neither "within the U.S." nor "overseas."

All applicable requirements should be specified. This is the toughest condition to be checked. There is really no way to be sure that all the requirements are captured and that one week before the production date one of the stakeholders won't say, "I forgot to mention that I need one more feature in the application."

A good requirement should have more criteria. However, they usually can be expressed as a combination of the criteria we have just discussed:

- Modifiable: If it is atomic and nonredundant, it is usually modifiable.
- Traceable: If it is atomic and has a unique ID, it is usually traceable.

1.5 An Overview of the Requirements Management Process

A simplified description of the requirements management process contains the following major steps:

- Establishing a requirements management plan
- Requirements elicitation
- Developing the Vision document
- Creating use cases
- Supplementary specification
- Creating test cases from use cases
- Creating test cases from the supplementary specification
- System design

The first step (requirements management plan) defines the requirements pyramid. In each of the next seven steps, one of the elements of the pyramid is built. Table 1.1 describes which requirement types and what documents are created in each step. As you can see, the process navigates through the pyramid from the top down and from left to right.

Table 1.1 Requirements and Documents Created in Each Phase

Step	Requirement Types	Documents
Requirements elicitation	Stakeholder needs	Stakeholder requests
Developing the Vision document	Features	Vision
Creating use cases	Use cases, scenarios	Use case specifications
Supplementary specification	Supplementary requirements	Supplementary specification
Creating test cases from use cases	Test cases	Test cases
Creating test cases from the supplementary specification	Test cases	Test cases
System design	Class diagrams, interaction diagrams	UML diagrams

Requirements management is an interactive process. In a typical iteration, a full pass through the pyramid is performed. Even in the same iteration, we can go back a few steps and repeat the activity. For example, during the creation of a test case, we can discover that some information is missing, and we need more input from a stakeholder, so we go back a step to "gathering requirements." To maintain the model's integrity, it is important to update all affected requirements. In initial iterations the emphasis is placed on the first few steps (the top of the pyramid), and in later iterations more time is spent on the lower part of the pyramid.

This description is simplified because only major steps are described. For example, some activities defined by the Rational Unified Process (RUP) are more granular. (For example, our step of creating use cases contains the following RUP activities: find actors and use cases, structure the use case model, prioritize use cases, and detail use cases.)

Let's briefly look through all the steps.

Establishing a Requirements Management Plan

One of the first tasks in the project is developing a Requirements Management Plan (RMP). The RMP describes the overall approach to managing requirements in the project. The document details how requirements are created, organized, modified, and traced during the project lifecycle. It also describes all requirement types and their attributes used in the project.

Here are some questions that can be answered in the RMP:

- Will any RM tool be used?
- What requirement types will be tracked in the project?
- What are the attributes of these requirements?

- Where will the requirements be created—in the database only or in the documents?
- Between which requirements do we need to implement traceability?
- What documents are required?
- Which requirements and documents will be used as a contract with customers?
- If part of the project is outsourced, what requirements and documents will be used as a contract with a vendor?
- Will we follow the RUP or some other methodology?
- Does the customer need any specific documents to comply with his development process?
- How will change management be implemented?
- Assuming that RequisitePro is used, will the whole system be stored in one RequisitePro project or spread among many projects?
- What process will guarantee that all requirements were implemented and tested?
- Which requirements or views do we need to generate reports?

Chapter 3, "Establishing a Requirements Management Plan," describes all these decisions in detail.

Requirements Elicitation

At the top level of the pyramid are stakeholder needs, as shown in Figure 1.3.

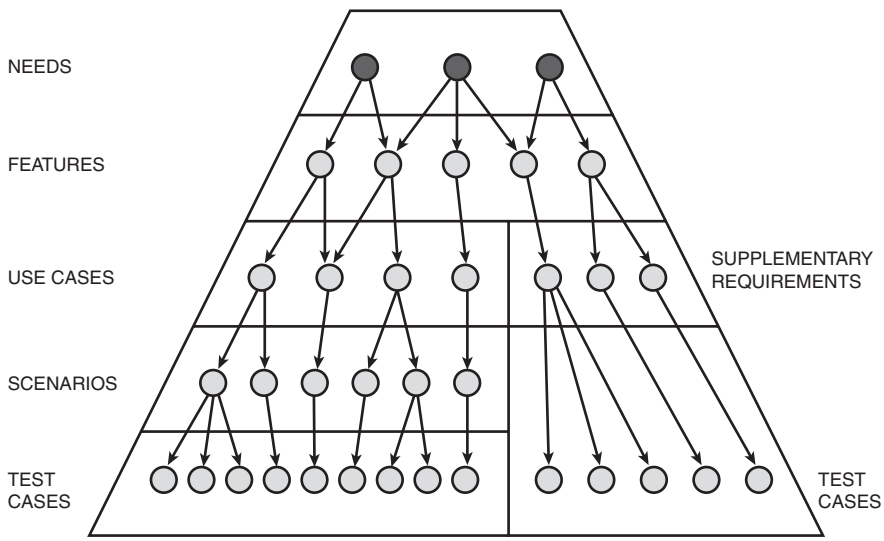

Figure 1.3 Needs (stakeholder requests) are at the top of the pyramid.

This book uses the terms "stakeholder needs" and "stakeholder requests" as synonyms. However, if project specifics require, it is possible to define them as two separate requirement types. In that case, needs would be very high-level requirements, such as "The system shall have the capability to book a flight." Usually there are no more than five high-level needs per stakeholder and no more than 15 needs per project. All detailed requirements would be captured as stakeholder requests. However, in many projects it is easier to capture all input from the stakeholders in the same type of requirement, so in the example used in this book, stakeholder needs represent all input from the stakeholders, regardless of granularity. In some projects there may be a need to distinguish between "stakeholder needs" describing initial requirements and "stakeholder requests" that may include subsequent change requests.

Requirements elicitation, also called requirements gathering, is a very important step. Missing or misinterpreting a requirement at this stage will propagate the problem through the development lifecycle.

Here are some of the techniques used to elicit requirements from stakeholders:

- Interviews
- Questionnaires
- Workshops
- Storyboards
- Role-playing
- Brainstorming sessions
- Affinity diagrams
- Prototyping
- Analysis of existing documents
- Use cases
- Analysis of existing systems

These techniques are described in Chapter 5, "Requirements Elicitation."

Developing the Vision Document

Section 1.3 discussed attributes of a good requirement. However, information that comes from stakeholders does not necessarily have these attributes. It is especially the case that requirements coming from different sources may be conflicting or redundant.

During development of the Vision document, one of the main goals of business analysts is deriving features from stakeholder needs (see Figure 1.4). Features should have all the attributes of a good requirement. They should be testable, nonredundant, clear, and so on.

The Vision document should contain essential information about the system being developed. Besides listing all the features, it should contain a product overview, a user description, a summary of the system's capabilities, and other information that may be required to understand the system's purpose. It may also list all stakeholder needs in case they were not captured in separate documents.

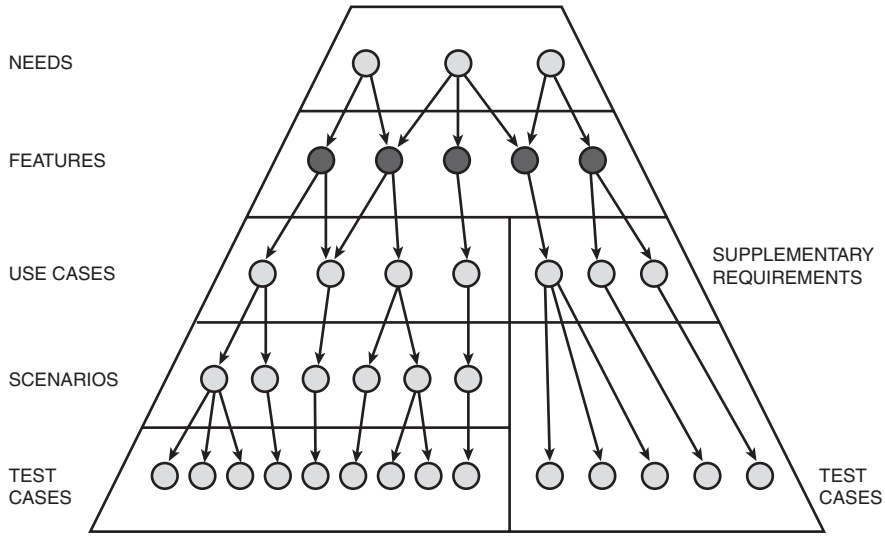

Figure 1.4 Features are derived from needs.

Some parts of the Vision are created at the very beginning, before we even start eliciting requirements from stakeholders. Examples of these sections are

- A description of the problem being solved.
- Identification of users/customers/stakeholders.

Creating Use Cases

Functional requirements are best described in the form of use cases. They are derived from features, as shown in Figure 1.5.

A use case is a description of a system in terms of a sequence of actions. It should yield an observable result or value for the actor (an actor is someone or something that interacts with the system). The use cases

- Are initiated by an actor.
- Model an interaction between an actor and the system.
- Describe a sequence of actions.
- Capture functional requirements.
- Should provide some value to an actor.
- Represent a complete and meaningful flow of events.

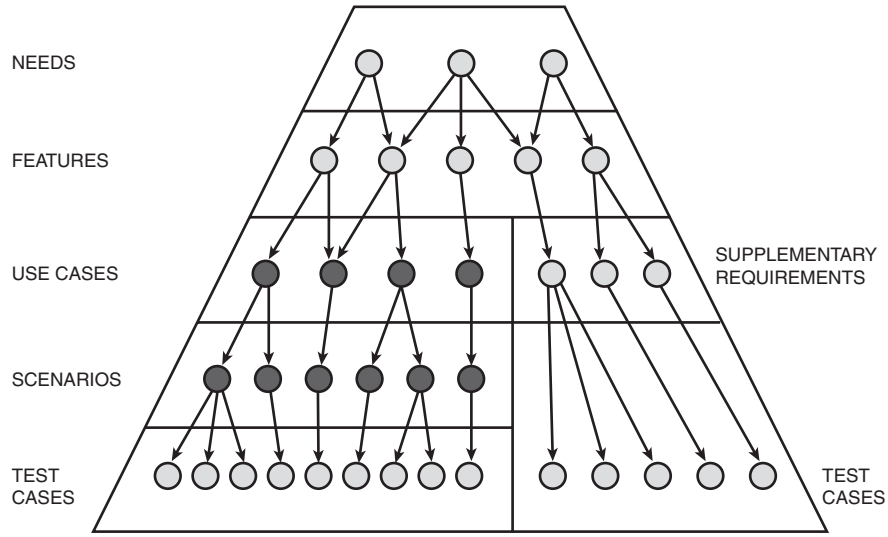

Figure 1.5 Use cases are derived from features that describe the system's functionality. Scenarios are derived from use cases.

Here is a fragment of a use case:

1. The user enters required flight information: departure airport and date, arrival airport and date.
2. The system displays all outbound flights matching the search criteria.
3. The user selects an outbound flight.
4. The system displays a list of available return flights.

The purpose of a use case is to facilitate agreement between developers, customers, and users about what the system should do. A use case becomes sort of a contract between developers and customers. It's also a basis for use case realizations, which play a major role in design. In addition, you can produce sequence diagrams, communication diagrams, and class diagrams from use cases. Furthermore, you can derive user documentation from use cases. Use cases may also be useful in planning the technical content of iterations and give system developers a better understanding of the system's purpose. Finally, you can use them as an input for test cases.

While designing use cases we will also define scenarios—specific paths through the use case. We usually implement systems scenario by scenario, not the whole use case at once. Scenarios are required when we derive test cases from use cases. On the requirements pyramid, scenarios are one level below use cases (see Figure 1.5).

Supplementary Specification

Supplementary specification captures nonfunctional requirements (usability, reliability, performance, supportability) and some functional requirements that are spread across the system, so it is tough to capture them in the use cases. These requirements are called supplementary requirements and are derived from features, as shown in Figure 1.6.

Chapter 8, "Supplementary Specification," discusses this type of requirement in detail.

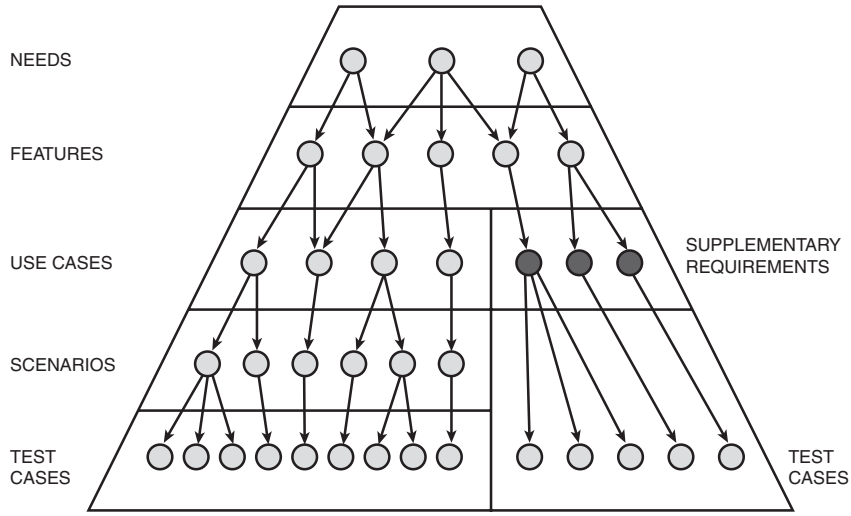

Figure 1.6 Supplementary requirements are derived from features that cannot be captured in the use cases.

Creating Test Cases from Use Cases

As soon as all the requirements are captured, we should design a way to check whether they are properly implemented in the final product. Test cases will show the testers what steps should be performed to test all requirements. In this step we will concentrate on creating test cases from use cases. If we did not create scenarios while generating use cases, we need to define them now. Test cases are at the lowest level of the pyramid, as shown in Figure 1.7.

This process is described in detail in Chapter 9, "Creating Test Cases from Use Cases."

Creating Test Cases from the Supplementary Specification

The approach used in the preceding step does not apply to testing supplementary requirements. Because these requirements are not expressed as a sequence of actions, the concept of scenarios does not apply to them. An individual approach should be applied to each of the supplementary requirements because techniques used to test performance requirements are different from techniques used to test usability requirements. In this step we also design testing infrastructure and platform-related issues.

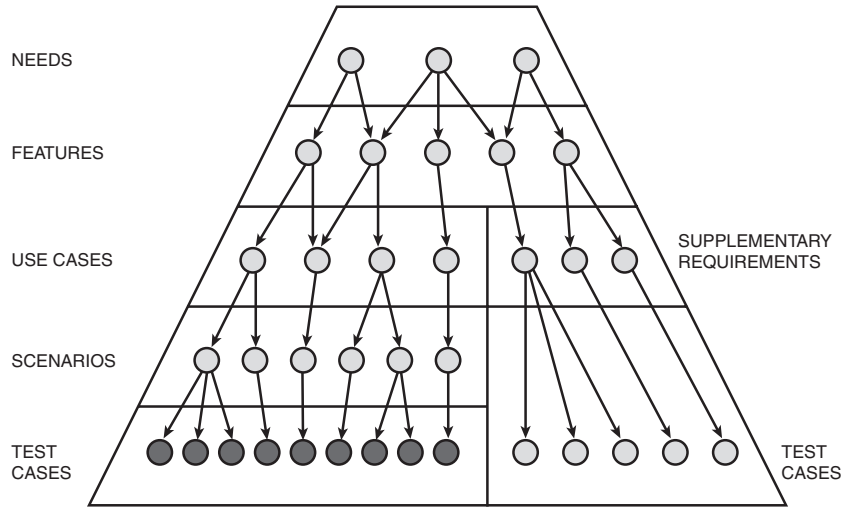

Figure 1.7 Test cases for testing use cases.

Sometimes we need to "borrow" one scenario that was created to test use cases (see Figure 1.8). For example, to test the requirement "The system should run using the Internet Explorer (IE) browser and using the Netscape browser," we should select one scenario (preferably basic flow of the most popular use case) and test the full scenario in the IE browser. Then we should test the same scenario again in the Netscape browser. There is no need to test all test cases created in the preceding step in both browsers. Just select those that contain some functionality that may be browser-specific.

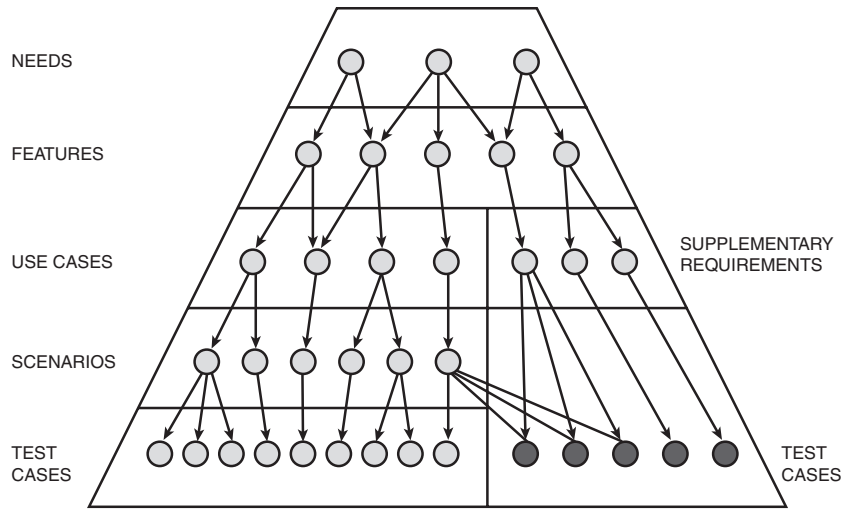

Figure 1.8 Test cases for testing supplementary requirements.

Some of the supplementary requirements can be tested using automated testing tools such as Rational Robot.

System Design

Requirements are the basis for system design, which is often facilitated by use of the Unified Modeling Language (UML) [BOO98]. Many tools, such as Rational Rose, Rational Software Architect, Rational Data Architect, and Rational Software Modeler, can significantly facilitate the creation of all required diagrams.

One approach is to create interaction diagrams from scenarios and, at the same time, assign functionality to the classes (see Figure 1.9). This topic is discussed in detail in Chapter 11, "Object-Oriented Design."

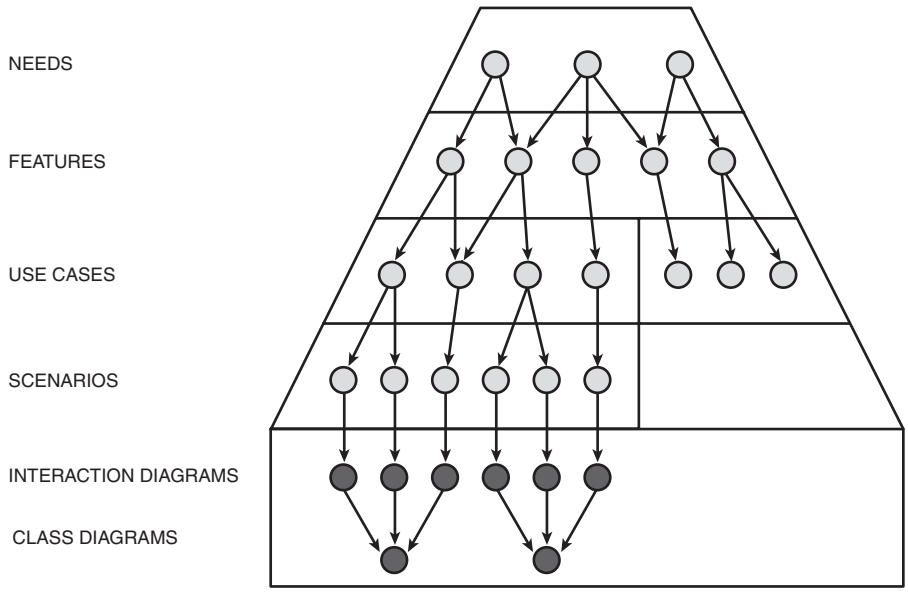

Figure 1.9 System design.

1.6 Summary

This chapter presented the RM process from the point of view of created requirements and documents. In short, this is the approach:

1. Stakeholder needs are gathered and documented in stakeholder requests documents.
2. Features are derived from needs and are documented in Vision.

3. Use cases and supplementary requirements are derived from features.
4. Test cases are derived from use cases and supplementary requirements.

These steps should be applied iteratively during the project lifecycle.

References

[BOO98] Booch, Grady, James Rumbaugh, and Ivar Jacobson. *UML User Guide*, Boston, MA: Addison-Wesley, 1998.

[HUL05] Hull, Elizabeth, Kenneth Jackson, and Jeremy Dick. *Requirements Engineering*, London: Springer, 2005.

[LEF03] Leffingwell, Dean, and Don Widrig. *Managing Software Requirements: A Use Case Approach,* Second Edition, Boston, MA: Addison-Wesley, 2003.

[LUD05] Ludwig Consulting Services, LLC, www.jiludwig.com.

[YOU01] Young, Ralph R. *Effective Requirements Practices*, Boston, MA: Addison-Wesley, 2001.